ARO 15019.1-A-EL

FOR FURTHER TRAN [illegible]

AD-A055599

1

ISP Descriptions of Four Military Computer Architectures

April 1978

Department of Computer Science
Carnegie -Mellon University
Pittsburgh, Pennsylvania

This work was supported by the Army Research Office under contract number DAAG29-77-C-0033.

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER 15019.1-A-EL	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ISP Descriptions of Four Military Computer Architectures		5. TYPE OF REPORT & PERIOD COVERED Final Report: 15 Aug 77 - [illegible] Dec 77
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) M. R. Barbacci		8. CONTRACT OR GRANT NUMBER(s) DAAG29 77 C [illegible]83
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P. O. Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE April 1978
		13. NUMBER OF PAGES 200 (approx)
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

[illegible]

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

[illegible]

DD FORM 1473

20. ABSTRACT CONTINUED

This report consists of the four [illegible] descriptions. The report of the results of the companion evaluation entitled "Phase II Comparative Evaluation of MCF Computer Architectures" is appended to this report.

X

A 23 0 8

1. Introducion

This is the final technical report for Contract DAAG29-77-C-033. The purpose of the contract was to support a companion contract DAAG29-77-C-034 in an evaluation of alternative military architectures. In order to evaluate the computer architectures, an ISP description must be constructed for each architecture. This description drives a simulator that is used to debug and measure the test programs written for the evaluation. Under this contract, the ISP descriptions for four military computers, the AN/UYK-7, the AN/GYK-12, the AN/UYK-19 and the AN/UYK-20, were constructed. This report consists of the four ISP descriptions. The report of the results of the companion evaluation entitled "Phase II Comparative Evaluation of MCF Computer Architectures" is appended to this report.

2. AN/UYK-7 ISPL Description

```
! AN/UYK-7 ISP DESCRIPTION
!
!                       TABLE OF CONTENTS
!
!       ROUTINE                                             PAGE
!
!       INTRODUCTION                                          2
!       CHANGE HISTORY                                        2
!       MACRO DEFINITIONS                                     3
!       MEMORY STATE DEFINITIONS                              3
!       PROCESSOR STATE DEFINITIONS                           4
!       INSTRUCTION REGISTER DEFINITIONS                      6
!       ISP IMPLEMENTATION RELATED VARIABLES                  7
!       UTILITY ROUTINES                                      9
!       INSTRUCTION ADDRESS GENERATION                       15
!       OPERAND ADDRESS CALCULATION                          16
!       OPERAND AND CHARACTER ADDRESS ROUTINES               17
!       READ OPERAND AND WRITE OPERAND                       18
!       READ DOUBLE LENGTH VARIABLES                         19
!       READ FORMAT I OPERAND                                19
!       WRITE FORMAT I OPERAND                               20
!       CHECK REPEAT TERMINATION                             21
!       INTERRUPT HANDLER                                    22
!       CONDITION CODE ROUTINES                              24
!       FORMAT I INSTRUCTION EXECUTION                       25
!       FORMAT I INSTRUCTION DECODE TABLE                    36
!       FORMAT II INSTRUCTION EXECUTION                      38
!       FLOATING POINT INSTRUCTIONS                          48
!       FORMAT II INSTRUCTION DECODE TABLE                   50
!       FORMAT III INSTRUCTION EXECUTION                     51
!       FORMAT III INSTRUCTION DECODE TABLE                  58
!       FORMAT IV INSTRUCTION EXECUTION                      60
!       FORMAT IV INSTRUCTION DECODE TABLE                   71
!       INSTRUCTION EXECUTION                                73
!       INSTRUCTION CYCLE                                    73
!
!       MAIN EXECUTABLE PROGRAM                              73
! ISP description of the AN/UYK-7 computer architecture.

! The AN/UYK-7 is manufactured by the Univac Division of the
! Sperry Rand Corporation.

! The AN/UYK-7 is billed to be a "highly reliable ruggedized
! multiple processor system designed ...... for military
! applications".

!       The AN/UYK-7 architecture presented here was coded in accordance
!       with the "AN/UYK-7 Technical Description" manual, Sperry-Univac,
!       Revised May, 1971.

!       G.W.LEIVE
!       KONRAD LAI
!       CARNEGIE-MELLON UNIVERSITY
!       PITTSBURGH, PENNSYLVANIA 15213

!       V1.7
!       28 JUL 1977

!       V1.7 FOR FORMAT III INSTRUCTION, BOTH PS AND PD ARE LOADED DURING
!       AN JUMP.  PS FROM THE S FIELD, PD FROM THE Y + (B(B)) OF THE LAST
!       ADDRESS FETCH.  28 JUL 77,KKL

!       V1.6 FIXED WRITOP TO CORRECTLY STORE ON INDIRECT ADDRESSING.
!       PROBLEM WAS DISCOVERED FOR DOUBLE STORE, BUT SHOULD
!       HAVE EFFECTED ALL INDIRECT STORES (NOT FORMAT I).
!       24 JUN 77, GWL.

!       V1.5 16BIT ONE'S COMPLEMENT INDEX ADDER IS USED TO GENERATE
!       PARTIAL ADDRESS & FORMS LITERAL OPER[illegible] .S (P21).  INDEX REGISTER
!       IS CONSIDERED AS UNSIGNED 16BIT QUANTITY FOR B7 IN REPEAT.

!       V1.4 CHANGES ATTEMPT TO CORRECT   [illegible]THER "GOTCHA"
!       IN THE REPEAT INSTRUCTION.  (RF       OF THE MANUAL)
!       REPEATED SEQUENTIAL CHARACTER      [illegible]SSING ACTS
!       LIKE SINGLE CHARACTER ADDRESSING UNLESS THE REPEATD
!       INSTRUCTION TERMINATES OR IS INTERRUPTED.  IN THOSE
!       CASES, THE ICW IS UPDATED.
!       UNRELATED TO ABOVE, C FIELD CODES FOR WORD AND
!       SINGLE CHARACTER INDIRECT ADDRESSING WERE REVERSED.
!       (REF P33 OF THE MANUAL).

!       V1.3 REPEAT INDEX INCREMENT (REF P54 OF THE MANUAL)
!       WERE INADVERTENTLY EXCLUDED FROM EARLIER VERSIONS OF
!       THIS DESCRIPTION.
! MACRO DEFINITIONS


ANUYK7:=

        !DECLARE

        MACRO BEGIN:=( $
        MACRO END:=) $
        MACRO MAXW:=32767 $              !ARCHITECTURE SUPPORTS 262,144 WORDS
        MACRO MAXNDR:=511 $              !AND 512 WORDS OF NDRO
        MACRO NO.OP:=T0+T0 $             !NO-OPERATION
        MACRO ONE32:=#37777777777 $      !NEGATIVE ZERO

! MEMORY STATE

! PRIMARY MEMORY

        MW[0:MAXW]<31:0>;                !PRIMARY WORD MEMORY

! NON-DESTRUCTIVE READ-OUT (NDRO) MEMORY
! MAGNETIC CORE ROPE MEMORY WHICH NORMALLY CONTAINS:

!       THE HARDWARE INTERRUPT ANALYSIS ROUTINE
!       TWO INITIAL LOAD OR AUTOMATIC RECOVERY ROUTINES (BOOTSTRAP).
!       A DIAGNOSTIC PROGRAM

        NDRO[0:MAXNDR]<31:0>;


! I/O CONTROLLER INTERFACE

        IOC[0:3]<31:0>;                  ! WRITE ONLY (BY IO INSTRUCTION)
```

```
! PROCESSOR STATE

! OPERATIONAL REGISTERS

        P<19:0>;                        ! PROGRAM ADDRESS REGISTER
            PS<2:0>:=P<19:17>;          ! P REGISTER S (BASE) FIELD
            PD<15:0>:=P<15:0>;          ! P REGISTER D (DISPLACEMENT)

! CPU CONTROL MEMORY

        CMR[#0:#177]<31:0>;             ! CPU CONTROL MEMORY

!    TASK MODE

        ACT[#0:#7]<31:0>:=CMR[#0:#7]<31:0>;     ! ACCUMULATORS 0-7

        RBT[#0:#7]<31:0>:=CMR[#10:#17]<31:0>;   ! INDEX (B) REGISTERS
                                                ! RBT[0] IS UNASSIGNED -
                                                ! INCLUDED FOR ISPL

        RST[#0:#7]<31:0>:=CMR[#20:#27]<31:0>;   ! BASE (S) REGISTERS

! BREAKPOINT REGISTER

        BPR<19:0>:=CMR[#60]<19:0>;      ! BREAK POINT REGISTER
            BPCODE<1:0>:=BPR<19:18>;    ! SELECTION CRITERIA
                                        !   0 => DISABLED
                                        !   1 => INSTRUCTION ADDRESS
                                        !   2 => OPERAND ADDRESS
                                        !   3 => INST AND OPND ADDRESS

            BPADDR<17:0>:=BPR<17:0>;    ! COMPARISON ADDRESS

! ACTIVE STATUS REGISTER

        ASR<22:0>:=CMR[#70]<22:0>;      ! ACTIVE STATUS REGISTER

            CPUID<2:0>:=ASR<22:20>;     ! HARDWIRED CPU IDENTIFIER
            UPLOW<>:=ASR<15>;           ! SET WHEN UPPER HALFWORD
                                        !  INSTRUCTION HAS BEEN EXECUTED
                                        !  CLEARED OTHERWISE
            CLASLO<1:3>:=ASR<14:12>;    ! LOCK OUT LOWER PRIORITY INTS.
            BASE<>:=ASR<11>;            ! BASE (S) REGISTER SELECT
            AISEL<>:=ASR<10>;           ! [illegible]INDEX (B) REGISTER SELECT
            MLD<>:=ASR<9>;              ! MEMORY LOCKOUT DISABLE
            LBENA<>:=ASR<8>;            ! LOAD BASE ENABLE
            BSMODE<>:=ASR<7>;           ! BOOTSTRAP MODE
            SPARE<2:0>:=ASR<6:4>;       ! PROGRAMMABLE SPARE BITS
            CC<3:0>:=ASR<3:0>;          ! CONDITION CODES
                FIXOV<>:=ASR<3>;        ! FIXED POINT OVERFLOW
                CCEQL<>:=CC<2>;         !   1 => EQUAL
                CCGEQ<>:=CC<1>;         !   1 => GREATER OR EQUAL
                CCLIM<>:=CC<0>;         !   1 => OUT OF LIMITS
```

```
! PROCESSOR STATE (PAGE 2)

! CPU CONTROL MEMORY

!     INTERRUPT MODE

        AC1[#0:#7]<31:0>:=CMR[#100:#107]<31:0>; !ACCUMULATORS 0-7

        CPMCLK<18:0>:=CMR[#110]<18:0>;          !CPU MONITOR CLOCK REGISTER

        PB1[#0:#7]<31:0>:=CMR[#110:#117]<31:0>; !INDEX (B) REGISTERS

        PS1[#0:#7]<31:0>:=CMR[#120:#127]<31:0>; !BASE (S) REGISTERS

! STORAGE PROTECTION REGISTERS

        SPR[0:7]<31:0>:=CMR[#160:#167]<31:0>;   !STORAGE PROTECTION REGISTERS

! SEGMENT IDENTIFICATION REGISTERS

        SIR[0:7]<31:0>:=CMR[#170:#177]<31:0>;   !SEGMENT IDENTIFICATION REGS
```

```
! INSTRUCTION REGISTER (U) OPERAND FIELDS

        U<31:0>;                         !INSTRUCTION REGISTER

!           FIELD DEFINITIONS            FIELD   FORMAT

            A<2:0>:=U<25:23>;            ! A      I, II, III, IV(A,B)
            AF4<5:0>:=U<25:20>;          ! COMBINED AF4 FIELD
            AK<5:0>:=U<25:20>;           ! COMBINED AK FIELD
            B<2:0>:=U<19:17>;            ! B      I, II, III, IV(A)
            F<5:0>:=U<31:26>;            ! F      I, II, III, IV(A,B)
                F0<2:0>:=F<5:3>;         ! UPPER HALF OF F
                F1<2:0>:=F<2:0>;         ! LOWER HALF OF F

            F2<2:0>:=U<22:20>;           ! F2     II
            F3<1:0>:=U<22:21>;           ! F3     III
            F4<2:0>:=U<22:20>;           ! F4     .V(A)
            I<>:=U<16>;                  ! I      I, II, III, IV(A)
            K<2:0>:=U<22:20>;            ! K      I
            K2<>:=U<20>;                 ! K      III
            M<6:0>:=U<22:16>;            ! M      IV(B)
            S<2:0>:=U<15:13>;            ! S      I, II, III
            SY<15:0>:=U<15:0>;           ! SY     I, II, III
            Y<12:0>:=U<12:0>;            ! Y      I, II, III

            UBISY<19:0>:=U<19:0>;        ! U REGISTER B, I, S, Y FIELDS

            UHI<15:0>:=U<31:16>;         ! UPPER HALFWORD
            ULO<15:0>:=U<15:0>;          ! LOWER HALFWORD

! V REGISTER AND ICW FORMAT

!       THE V REGISTER IS MENTIONED IN THE AN/UYK-7 TECHNICAL DESCRIPTION,
!       BUT ITS USE WAS NOT SPECIFIED.  IT WAS CHOSEN FOR USE WHEN
!       INTERPRETING INDIRECT CONTROL WORD (ICW) FORMATS.

        V<31:0>;                         ! V-INTERNAL DECODE REGISTER

            C<1:0>:=V<31:30>;            ! CONTROL DESIGNATOR
            CI<>:=V<29>;                 ! IND. SUBFUNCTION DESIGNATOR
            D<15:0>:=V<15:0>;            ! ADDRESS DISPLACEMENT
            POS<4:0>:=V<24:20>;          ! POSITION INDICATOR
            VBISY<19:0>:=V<19:0>;        ! B, I, S, Y FIELDS
            VY<12:0>:=V<12:0>;           ! Y FIELD
            W<4:0>:=V<29:25>;            ! CHARACTER LENGTH DESIGNATOR
```

```
! ISP IMPLEMENTATION RELATED VARIABLES

        MIAR<17:0>;                      ! INSTRUCTION ADDRESS REGISTER
        MIBR<31:0>;                      ! INSTRUCTION BUFFER REGISTER
            MIBRHI<15:0>:=MIBR<31:16>;   ! UPPER HALFWORD IN MIBR
            MIBRLO<15:0>:=MIBR<15:0>;    ! LOWER HALFWORD IN MIBR

        MOAR<17:0>;                      ! OPERAND ADDRESS REGISTER
        MOBR<31:0>;                      ! OPERAND BUFFER REGISTER
        MOAR1<17:0>;                     ! TEMPORARY ADDRESS BUFFER
        MOBR1<31:0>;                     ! TEMPORARY OPERAND BUFFER

        MASK<31:0>;                      ! MASK FOR CHARACTER INSERT

        T0<0>;                           ! NO-OP REGISTER

        TAC<32:0>;                       ! TEMPORARY ACCUMULATOR
        TA0<31:0>;                       ! TEMPORARY ACCUMULATOR
        TA1<31:0>;                       ! TEMPORARY ACCUMULATOR
        TA2<31:0>;                       ! TEMPORARY ACCUMULATOR

        TDAC<64:0>;                      ! TEMPORARY DOUBLE ACCUMULATOR

        TD0<63:0>;                       ! TEMPORARY DOUBLE ACCUMULATOR
        TD1<63:0>;                       ! TEMPORARY DOUBLE ACCUMULATOR
        TD2<63:0>;                       ! TEMPORARY DOUBLE ACCUMULATOR

        TB<19:0>;                        ! TEMPORARY INDEX REGISTER
            TBS<2:0>:=TB<19:17>;
            TBD<15:0>:=TB<15:0>;
        TB1<16:0>;                       ! TEMPORARY REGISTER OF INDEX ADD

        TS<17:0>;                        ! TEMPORARY BASE REGISTER

        LASTAD<17:0>;                    ! TEMP FOR LAST OPERAND ADDRESS
                                         !   USED FOR UPDATING ICW III
        ILOCK<>;                         ! SPECIAL INTERRUPT LOCKOUT

        ISC<15:0>;                       ! INTERRUPT STATUS CODE
        INTVEC<1:4>;                     ! INTERRUPT CLASS VECTOR

        POWER<>;                         ! POWER FAIL FLAG (1 => FAILURE)

        AUTO.REC<>;                      ! FRONT PANEL SWITCH

        BOOTSTRAP<1:0>;                  ! BOOTSTRAP SW. (3 POS: 0, 1, 2)

        AUTO.START<>;                    ! FRONT PANEL SWITCH

        STOPBIT<>;                       ! STOP SWITCH
```

```
! ISP IMPLEMENTATION RELATED VARIABLES (PAGE 2)

        AA<2:0>;                        ! GENERAL ACCUMULATOR REG ADDR
        BA<2:0>;                        ! GENERAL INDEX REG ADDRESS
        SA<2:0>;                        ! GENERAL BASE REG ADDRESS

        IMS<>;                          ! INTERRUPT MODE BASE REGS

        COUNT<5:0>;                     ! JUNK COUNTER

        IFLAG<>;                        ! INDIRECT FLAG

        EXRF<>;                         ! EXECUTE REMOTE FLAG

        RPFLAG<>;                       ! REPEAT FLAG
        RCC<2:0>;                       ! REPEAT CONDITION CODES

        WTFLAG<>;                       ! WRITE TO MEMORY FLAG
        COMPAR<>;                       ! COMPAR INSTRUCTION INDICATOR
        REPLAC<>;                       ! REPLACE INDICATOR

        SIGN<>;                         ! SIGN HOLDER FOR ARITHMETIC OPS
        SIGN1<>;                        ! SIGN HOLDER FOR ARITHMETIC OPS

        JUMPSW<2:0>;                    ! JUMP SWITCH

        STOPSW<2:0>;                    ! STOP SWITCH

        HWFLAG<>;                       ! FLAG FOR HWFI EXECUTION

        NOP<1:0>;                       ! NORO FLAGS FOR INTERRUPT
                                        !  CLASSES I AND II

        RPTA<2:0>;                      ! SPECIAL "A" FIELD FOR REPEAT INSTRUCTION
        RPTB<2:0>;                      ! SPECIAL "B" FIELD FOR REPEAT
        RPTSY<15:0>;                    ! SPECIAL "SY" FIELD FOR REPEAT

        TSTR<1:0>;                      ! HOLDS TST RESULT FOR COMPARES

        SHCOUNT<31:0>;                  ! SHIFT COUNT COUNTER
```

```
! UTILITY ROUTINES

!       GET BASE REGISTER

        GETS:=  BEGIN
                DECODE BASE =>
                \0  TS ← RS1[S0]<17:0>;
           #    \1  TS ← RS1[S0]<17:0>
                END;

!       CHECK OPERAND READ

        CKOPRD:=BEGIN
                IF NOT MLD =>
                    BEGIN
                    IF NOT SPR[S0]<19> =>
                        (INTVEC<2> ← 1;
                        ISC ← #6 NEXT
                        BAILOUT ICYCLE
                        )
                    END
                END;    !END CKOPRD

!       CHECK OPERAND ADDRESS LIMIT

        CKOPAD:=BEGIN
                IF NOT MLD =>
                    S0 ← 5 NEXT
                    GETS NEXT           ! BASE REGISTER RETURNS IN "TS"
                    (IF (MOAR GTR (TS + SPR[S0]<15:0>)) =>
                        INTVEC<2> ← 1;
                        ISC ← #12
                    )
                END;    !END CKOPAD

!       CHECK INSTRUCTION BREAKPOINT

        CKIBPT:=BEGIN
                IF BPCODE<0> =>
                    (IF MIAR EQL BPADDR =>
        BREAK:=         (
                        INTVEC<2> ← 1;
                        ISC ← #13
                        )
                    )
                END;    !END CKIBPT

!       CHECK OPERAND BREAKPOINT

        CKOBPT:=BEGIN
                IF BPCODE<1> =>
                    (IF MOAR EQL BPADDR =>
                     INTVEC<2> ← 1;
                     ISC ← #5)
                END;    !END CKOBPT
```

```
! UTILITY ROUTINES (PAGE 2)

!       THESE ROUTINES ARE USED TO SELECT EITHER THE MAIN MEMORY
!       OR THE NDRO FOR INSTRUCTION READ.
!       OPERAND READ IS ALWAYS FROM THE MAIN MEMORY.
!       THE NDRO IS USED UNDER CERTAIN INTERRUPT CONDITIONS.
!       A REAL AN/UYK-7 WOULD CONTAIN TRUE READ ONLY ROUTINES.
!       THIS SIMULATION WOULD NEED TO HAVE ANY SPECIAL INTERRUPT
!       ROUTINES INSERTED (BY A SIMULATOR "READ" COMMAND) PRIOR
!       TO EXECUTION.

        MRI:=   ! MEMORY READ - INSTRUCTION
                BEGIN
                DECODE (NDR NEQ 0) =>
                \0      BEGIN
                        CKIBPT NEXT
                        MIBR ← MW[MIAR<17:0>]
                        END;
                \1      MIBR ← NDRO[MIAR<8:0>]
                END;    ! END MRI

        MRO:=   ! MEMORY READ - OPERAND
                BEGIN
                CKOPAD NEXT
                CKOPRD NEXT
                CKOBPT NEXT
                MOBR ← MW[MOAR<17:0>]
                END;    !END MRO

!       THE FOLLOWING ROUTINES ARE USED TO WRITE TO MAIN MEMORY.
!       THE WRITE FLAG IS USED TO CHECK TERMINATION OF REPEATED INSTRUCTIONS.

!       CHECK OPERAND WRITE

        CKOPWT:=BEGIN
                IF NOT MLO =>
                    BEGIN
                    IF NOT SPR[50]<18> =>
                        INTVEC<2>←1;
                        ISC ← #11 NEXT
                        BAILOUT ICYCLE
                    END
                END;    !END CKOPWT

        MWO:=   BEGIN
                CKOPAD NEXT
                CKOPWT NEXT
                CKOBPT NEXT
                MW[MOAR<17:0>] ← MOBR;
                WTFLAG ← 1
                END;    !END MWO
```

```
! UTILITY ROUTINES (PAGE 3)

!       GET ACCUMULATOR

        GETA0:= BEGIN
                DECODE AISEL =>
                \0  TA0 ← ACT[A0];
                \1  TA0 ← ACI[A0]
                END;

!       GET AC SPECIFIED BY A FIELD

        GETA:=  BEGIN
                A0 ← A NEXT
                GETA0
                END;

!       STORE ACCUMULATOR

        PUTA0:= BEGIN
                DECODE AISEL =>
                \0  ACT[A0] ← TA0;
                \1  ACI[A0] ← TA0
                END;

!       GET ACCUMULATOR [A0 + 1]

        GETA1:= BEGIN
                (IF (A0 + 1) GTR #7 =>
                    BEGIN
                    ASP<0> ← 1;
                    INTVEC<2> ← 1;
                    ISC ← #12 NEXT
                    BAILOUT ICYCLE
                    END
                ) NEXT
                (DECODE AISEL =>
                \0  TA1 ← ACT[(A0 + 1)<2:0>];
                \1  TA1 ← ACI[(A0 + 1)<2:0>]
                )
                END;    !END GETA1
```

```
! UTILITY ROUTINES (PAGE 4)

!       STORE ACCUMULATOR (A + 1)

        PUTA1:= BEGIN
                (IF (A0 + 1) GTR #7 =>
                    BEGIN
                    ASP<0> ← 1;
                    INTVEC<2> ← 1;
                    ISC ← #12 NEXT
                    BAILOUT ICYCLE
                    END
                ) NEXT
                (DECODE AISEL =>
                \0  ACT[(A0 + 1)<2:0>] ← TA1;
                \1  AC1[(A0 + 1)<2:0>] ← TA1
                )
                END;    !END PUTA1

!       GET INDEX REGISTER

        GETB:=  BEGIN
                (IF B0 EQL 0 => TB ← 0);
                (IF B0 NEQ 0 =>
                    (DECODE AISEL =>
                        \0  TB ← PBT[B0]<19:0>;
                        \1  TB ← RB1[B0]<19:0>
                    )
                )
                END;    !END GETB

!       STORE INDEX REGISTER

        PUTB:=  BEGIN
                (IF B0 NEQ 0 =>
                    (DECODE AISEL =>
                        \0  PBT[B0]<19:0> ← TB;
                        \1  RB1[B0]<19:0> ← TB
                    )
                )
                END;
```

```
! UTILITY ROUTINES (PAGE 5)

!       GET DOUBLE LENGTH VARIABLE (AC[A + 1], AC[A])

        GETD:=  BEGIN
                A0 ← A NEXT
                GETA0;
                GETA1 NEXT
                TD0<31:0> ← TA0;
                TD0<63:32> ← TA1
                END;    !END GETD

!       STORE DOUBLE LENGTH VARIABLE (AC[A + 1], AC[A])

        PUTD:=  BEGIN
                TA0 ← TD0<31:0>;
                TA1 ← TD0<63:32> NEXT
                PUTA0;                     ! STORE (TA0) IN ACCUMULATOR (A0)
                PUTA1                      ! STORE (TA1) IN AC(A0+1)
                END;    ! END PUTD

!       OPERATION EXCEPTION (ILLEGAL OP CODE)

        OPEX:=  BEGIN
                INTVEC<2> ← 1;  ISC ← #2
                END;    !END OPEX

!       CHECK PRIVILEGED INSTRUCTION

        CKPRIV:=BEGIN
                IF NOT ILOCK =>
                    BEGIN
                    IF ASR<19:16> EQL 0 =>
                        (INTVEC<2> ← 1; ISC ← #3 NEXT
                        BAILOUT ICYCLE)
                    END
                END;    !END CKPRIV

!       CHECK INDIRECT ADDRESSING

        CKIND:= BEGIN
                IF NOT MLO =>
                    (DECODE SPR[S0]<17> =>
                    \0  BEGIN
                        INTVEC<2> ← 1; ISC ← #6 NEXT
                        BAILOUT ICYCLE
                        END;
                    \1  (IF SPR[S0]<16> => IMS ← 1)
                    )
                    END;        !END CKIND
```

```
! UTILITY ROUTINES (PAGE 6)

!       PARITY GENERATOR

        PARITY:=BEGIN
                COUNT ← 31;
                T0 ← 0 NEXT
                WTPLP:= BEGIN
                        (IF TA0<0> => T0 ← (T0 + 1)<0>) NEXT
                        TA0 ← TA0 SRR 1;
                        COUNT ← (COUNT MINUS 1)<5:0> NEXT
                        (IF COUNT => WTPLP)
                        END
                END;    !END PARITY

!       DEVELOP THE SHIFT COUNT PER FIGURE 23 IN UYK-7 MANUAL

        SHIFTC:=BEGIN
                (DECODE M<6:5> =>
                \00     COUNT ← M<5:0>;
                \01     COUNT ← M<5:0>;
                \10     BEGIN
                        B0 ← B NEXT
                        GETB NEXT        ! INDEX REGISTER RETURNS IN "TB"
                        COUNT ← TB<5:0>
                        END;
                \11     BEGIN
                        A0 ← B NEXT
                        GETA0 NEXT
                        COUNT ← TA0<5:0>
                        END
                ) NEXT
                SHCOUNT ← (SHCOUNT + COUNT)<31:0> !*
                END;    !END SHIFTC

!       CHECK FLOATING POINT ERROR

        CKFPE:= BEGIN
                NO.OP
                END;    !END CKFPE
```

```
! INSTRUCTION ADDRESS GENERATION

        INSTAD:=BEGIN
                SA ← PS NEXT
                GETS NEXT                 ! BASE REGISTER RETURNS IN "TS"
                MIAR ← (PD + TS)<17:0> NEXT
                (IF NOT MLD =>
                    (IF (MIAR GTR (TS + SPR[PS]<15:0>)) =>
                        INTVEC<2> ← 1;          ! CLASS II INTERRUPT;
                        ISC ← #16 NEXT          ! INSTRUCTION LIMIT
                        BAILOUT ICYCLE
                    )
                )
                END;    !END INSTAD


! READ INSTRUCTION

        READIN:=BEGIN
                COMPAR ← 0;                     ! RESET COMPARE FLAG
                WTFLAG ← 0;                     ! RESET WRITE FLAG
                REPLAC ← 0;                     ! RESET REPLACE FLAG
                IFLAG ← 0 NEXT
                (IF UPLOW =>              ! HALF WORD INSTRUCTIONS
                    UHI ← ULO NEXT
                    BAILOUT READIN
                ) NEXT
                INSTAD NEXT
                MRI NEXT
                U ← MIBR NEXT
                (IF NOT MLD =>
                    (IF NOT SPR[PS]<20> =>
                        INTVEC<2> ← 1;          ! CLASS II INTERRUPT;
                        ISC ← #15 NEXT          ! INSTRUCTION EXECUTE
                        BAILOUT ICYCLE
                    )
                ) NEXT
                PD ← (PD + 1)<15:0>
                END;    !END READIN
```

```
! OPERAND ADDRESS CALCULATION

        OPAD:=  BEGIN
                (DECODE IFLAG =>)                    ! INDIRECT IN PROGRESS?
                \0  BEGIN                            ! NO. JUST A REGULAR ADDRESS
                    (DECODE (PPFLAG AND (PPTB NEQ 0) AND REPLAC) =>
                        S0 ← S;
                        S0 ← #6
                    ) NEXT
                    B0 ← B NEXT
                    GETB;                 ! INDEX REGISTER RETURNS IN "TB"
                    GETS NEXT             ! BASE REGISTER RETURNS IN "TS"
                    TB1 ← Y + TBD NEXT
                    TB1 ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                    MOAR ← (TB1 + TS)<17:0>
                    END;

                \1  BEGIN                            ! YES, INDIRECT.
                    (IF C EQL 0 =>)                  ! "C" FIELD OF ICW
                        BEGIN
                        DECODE C1 =>)                ! "C1" FIELD OF ICW
                        \0  BEGIN
                            S0 ← A NEXT
                            GETS NEXT    ! BASE REGISTER RETURNS IN "TS"
                            TB1 ← SY ;
                            MOAR ← (SY + TS)<15:0>
                            END;

                        \1  BEGIN
                            B0 ← B NEXT
                            GETB NEXT    ! INDEX REGISTER RETURNS IN "TB"
                            S0 ← TBS NEXT
                            GETS NEXT    ! BASE REGISTER RETURNS IN "TS"
                            TB1 ← SY + TBD NEXT
                            TB1 ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                            MOAR ← (TB1 + TS)<17:0>
                            END
                        END
                    );
                    (IF C NEQ 0 =>)
                        BEGIN
                        B0 ← B;
                        (DECODE (PPFLAG AND (PPTB NEQ 0) AND REPLAC) =>
                            S0 ← S;
                            S0 ← #6
                        ) NEXT
                        GETB;            ! INDEX REGISTER RETURNS IN "TB"
                        GETS NEXT        ! BASE REGISTER RETURNS IN "TS"
                        TB1 ← Y + TBD NEXT
                        TB1 ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                        MOAR ← (TB1 + TS)<17:0>
                        END
                    )
                    END
                )
                END;    !END OF OPAD
```

```
! OPERAND AND CHARACTER ADDRESS ROUTINES

        OPAD1:= BEGIN
                (DECODE I=>
                \0  OPAD;                    ! NOT INDIRECT
                \1  BEGIN                    ! INDIRECT
                    OPAD NEXT                ! ADDRESS OF ICW
                    MRD NEXT
                    V ← MOBR NEXT            ! ICW TO V REGISTER
                    IFLAG ← 1;
                    LASTAD ← MOAR;           ! SAVE ADDRESS FOR ICW UPDATE
                    UBISY ← VBISY NEXT
                    OPAD1
                    END
                )
                END;     !END OF OPAD1

        CHARAD:=BEGIN
                IF C<1> AND (NOT RPFLAG) =>
                    BEGIN
                    (DECODE (W GTR POS) =>
                    \0  POS ← (POS MINUS W)<4:0>;
                    \1  BEGIN
                        POS ← (32 MINUS W)<4:0>;
                        VY ← (VY + 1)<12:0>
                        END
                    );
                    UBISY ← VBISY;
                    MOAR1 ← MOAR;
                    MOBR1 ← MOBR NEXT
                    MOAR ← LASTAD;
                    MOBR ← V NEXT
                    MWD NEXT
                    MOAR ← MOAR1;
                    MOBR ← MOBR1
                    END
                END;     !END OF CHARAD
```

```
! READ OPERAND

          READOP: BEGIN
                   (IF NOT PPFLAG => IFLAG ← 0) NEXT
                   OPAD1 NEXT
                   (DECODE IFLAG =>
                   \0  MPD;
                   \1  BEGIN
                       (DECODE C<0> =>
                       \0  MRD;

                       \1  BEGIN
                           MASK ← 0 NEXT
                           MASK ← MASK (SL1 W NEXT
                           MRD NEXT
                           MOBR ← MOBR (SR0 POS NEXT
                           MOBR ← MOBR AND MASK NEXT
                           CHARAD
                           END
                       )
                       END
                   )
                   END;    !END OF READOP

! WRITE OPERAND

          WTCHAR:=BEGIN
                   MASK ← 0 NEXT
                   MASK ← MASK (SL1 W NEXT
                   MASK ← MASK (SL0 POS NEXT
                   MOBR1 ← MOBR1 (SL0 POS NEXT
                   MOBR ← (MOBR1 AND MASK) OR (MOBR AND (NOT MASK)) NEXT
                   MWD NEXT
                   CHARAD
                   END;

          WRITOP:=BEGIN
                   (IF NOT PPFLAG => IFLAG ← 0) NEXT
                   MOBR1 ← MOBR NEXT                             ! V1.6
                   OPAD1 NEXT
                   MOBR ← MOBR1 NEXT                             ! V1.6
                   (DECODE IFLAG =>
                       MWD;

                       (DECODE C<0> =>
                           MWD;

                           BEGIN
                           MOBR1 ← MOBR NEXT
                           MRD NEXT
                           WTCHAR
                           END)
                   )
                   END;    !END OF WRITOP
```

```
!       READ DOUBLE LENGTH VARIABLE

        READD:= BEGIN
                READOP NEXT
                TD1<31:0> ← MOBR;
                MOAR ← (MOAR + 1)<17:0> NEXT
                MPO NEXT
                TD1<63:32> ← MOBR
                END;    !END READD

!       READ FORMAT I OPERAND (OPERAND ENDS UP IN TA2)

        RDFMTI:=BEGIN
                (IF Y NEQ 0 => READOP) NEXT
                (DECODE K =>
                IMMED:= BEGIN
                        B0 ← B NEXT
                        GETB NEXT       ! INDEX REGISTER RETURNS IN "TB"
                        TB1 ← SY + TB0 NEXT
                        TA2 ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                        (IF TA2<15> => TA2<31:16> ← #177777)
                        END;
                RHALF0:=BEGIN
                        TA2 ← MOBR<15:0> NEXT
                        (IF TA2<15> => TA2<31:16> ← #177777)
                        END;
                RHALF1:=BEGIN
                        TA2 ← MOBR<31:16> NEXT
                        (IF TA2<15> => TA2<31:16> ← #177777)
                        END;
                RFULL :=TA2 ← MOBR;
                RBYTE0:=TA2 ← MOBR<7:0>;
                RBYTE1:=TA2 ← MOBR<15:8>;
                RBYTE2:=TA2 ← MOBR<23:16>;
                RBYTE3:=TA2 ← MOBR<31:24>
                )
                END;    !END RDFMTI
```

```
!        WRITE FORMAT I OPERAND (OPERAND ENTERS IN TA2)

         WTONE:= BEGIN
                 IF F NEQ 0 =>
                     MPO NEXT
                     (DECODE F =>
                             NO.OP;
                     WHALF0:=MOBP<15:0> ← TA2<15:0>;
                     WHALF1:=MOBP<31:16> ← TA2<15:0>;
                     WFULL :=MOBP ← TA2;
                     WBYTE0:=MOBP<7:0> ← TA2<7:0>;
                     WBYTE1:=MOBP<15:8> ← TA2<7:0>;
                     WBYTE2:=MOBP<23:16> ← TA2<7:0>;
                     WBYTE3:=MOBP<31:24> ← TA2<7:0>) NEXT
                     MWO
                 END;

         RPFMTI:=BEGIN   !REPLACE FORMAT I
                 OPAD1 NEXT
                 (DECODE IFLAG =>
                     WTONE;
                     (DECODE C<0> =>
                         WTONE;
                         (MOBP1 ← TA2;
                         MPO NEX;
                         WTCHAR))
                 )
                 END;    !END WTFMTI

         WTFMTI:=BEGIN   !WRITE FORMAT I
                 (IF NOT RPFLAG => IFLAG ← 0) NEXT
                 RPFMTI
                 END;

         PUTBACK:=BEGIN
                 (IF RPFLAG =>
                     (IF IFLAG AND (C EQL 0) => IFLAG ← 0);
                     REPLAC ← 1
                 )
                 END;
```

```
! CHECK REPEAT TERMINATION

!       CHECK FOR REPEAT TERMINATION

        CKRPT := BEGIN
                B0 ← #7 NEXT                    ! COUNT IS IN B REG(7)
                GETB NEXT                       ! INDEX REGISTER RETURNS IN "TB"
                TB1 ← TBD - 1 NEXT              ! DECREMENT COUNT (UNSIGNED)
                TBD ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                PUTB NEXT                       ! STORE (TB) IN INDEX REGISTER B(B0)
                (IF (TBD EQL 0) OR (TBD EQL "FFFF) => RPFLAG ← 0);
                                                ! TERMINATE IF = 0
                B0 ← B NEXT                     ! INCREMENT OPERAND
                GETB NEXT                       ! ADDRESS INDEX REGISTER
                TB1 ← TBD + RPTSY NEXT          ! PER PAGE 54 OF
                                                ! THE AN/UYK-7 MANUAL
                TBD ← (TB1<16> + TB1<15:0>)<15:0> NEXT
                PUTB NEXT
                (DECODE COMPAR =>
                \0  BEGIN                       ! INSTRUCTION WAS NOT A COMPARE
                    DECODE RPTA =>              ! TERMINATE DEPENDS ON "A" FIELD
                    \0  (IF NOT RCC<2>     => RPFLAG ← 0);       ! NEQ 0
                    \1  (IF RCC<2>         => RPFLAG ← 0);       ! EQL 0
                    \2  (IF RCC<1>         => RPFLAG ← 0);       ! GEQ 0
                    \3  (IF NOT RCC<1>     => RPFLAG ← 0);       ! LSS 0
                    \4  NO.OP;
                    \5  (IF WTFLAG =>           ! IF WRITE TO MEMORY
                            (TA0 ← MOBR NEXT    ! AND IF EVEN PARITY,
                            PARITY NEXT         ! THEN TERMINATE REPEAT
                            (IF NOT T0     => RPFLAG ← 0)));
                    \6  (IF WTFLAG =>           ! IF WRITE TO MEMORY
                            (TA0 ← MOBR NEXT    ! AND IF ODD PARITY,
                            PARITY NEXT         ! THEN TERMINATE REPEAT
                            (IF T0         => RPFLAG ← 0)));
                    \7  NO.OP
                    END;

                \1  BEGIN       ! COMPARE INSTRUCTION BEING REPEATED
                    DECODE RPTA =>                      ! TERMINATE IF:
                    \0  (IF NOT CC<2>      => RPFLAG ← 0); ! NEQ 0
                    \1  (IF CC<2>          => RPFLAG ← 0); ! EQL
                    \2  (IF CC<2:1> EQL 1  => RPFLAG ← 0); ! GTR
                    \3  (IF CC<1>          => RPFLAG ← 0); ! GEQ
                    \4  (IF NOT CC<1>      => RPFLAG ← 0); ! LSS
                    \5  (IF CC<2:1> NEQ 1  => RPFLAG ← 0); ! LEQ
                    \6  (IF CC<0>          => RPFLAG ← 0); ! OUTSIDE LIMITS
                    \7  (IF NOT CC<0>      => RPFLAG ← 0)  ! WITHIN LIMITS
                    END
                ) NEXT
                (IF NOT RPFLAG => ILOCK ← 0;
                    (IF (C EQL 3) AND IFLAG => CHARAD))
                END;    !END CKRPT
```

```
! INTERRUPT HANDLER

        INTSET:=BEGIN
                HWFLAG ← 0;
                (IF PPFLAG =>
                    PPFLAG ← 0 NEXT
                    (IF C EQL 3 => CHARAD);
                    PD ← (PD MINUS 2)<15:0>) NEXT!*
                IFLAG ← 0
                END;

        INT:=   BEGIN                                       ! CLASS I
                (IF (INTVEC<1> AND (NOT CLASLO<1>)) =>
        ICS1:=      BEGIN
                    INTSET NEXT
                    CMP[#141] ← ASR<19:0> NEXT
                    CMP[#142] ← ISC;
                    CMP[#143] ← P;
                    ASR<19> ← 1;
                    UPLOW ← 0;
                    ASR<14:9> ← #77;
                    ASR<7> ← 1;
                    ASR<6:0> ← 0 NEXT
                    (DECODE POWLP =>
                    \0  (P ← NORD[0]<19:0>;
                         NDR<0> ← 1);
                    \1  (P ← CMR[#140]<19:0>));
                    POWER ← 0;
                    INTVEC<1> ← 0 NEXT
                    BAILOUT INT
                    END
                ) NEXT

                (IF (INTVEC<2> AND (NOT CLASLO<2>)) =>  ! CLASS II
        ICS2:=      BEGIN
                    INTSET NEXT
                    CMR[#145] ← ASR<19:0> NEXT
                    CMP[#146] ← ISC;
                    CMR[#147] ← P;
                    ASP<18> ← 1;
                    UPLOW ← 0;
                    ASP<13:9> ← #37;
                    ASP<7> ← 1;
                    ASR<6:0> ← 0 NEXT
                    (DECODE (AUTO.REC AND (ISC EQL #2)) =>
                    \0  P ← CMR[#144]<19:0>;

                    \1  (NDR<1> ← 1;
                        (DECODE BOOTSTRAP =>
                            P ← NORD[1]<19:0>;
                            P ← NORD[2]<19:0>;
                            P ← NORD[3]<19:0>;
                            NO.OP))
                    );
                    INTVEC<2> ← 0 NEXT
                    BAILOUT INT
                    END
                ) NEXT
```

```
! INTERRUPT HANDLER (PAGE 2)

                    ! CLASS III

          (IF ((INTVEC<3> AND (NOT CLASLO<3>))  =>
   ICS3:=      BEGIN
               INTSET NEXT
               CMP[#151] ← ASR<19:0> NEXT
               CMP[#152] ← ISC;
               CMP[#153] ← P;
               ASR<17> ← 1;
               UPLOW ← 0;
               ASR<12:9> ← #17;
               ASR<6:0> ← 0 NEXT
               P ← CMR[#150]<19:0>;
               INTVEC<3> ← 0 NEXT
               BAILOUT INT
               END
          ) NEXT

                    ! CLASS IV

          (IF INTVEC<4> =>
   ICS4:=      BEGIN
               INTSET NEXT
               CMP[#155] ← ASR<19:0> NEXT
               CMP[#156] ← ISC;
               CMP[#157] ← P;
               ASR<16> ← 1;
               UPLOW ← 0;
               ASR<11:9> ← #7;
               ASR<6:0> ← 0 NEXT
               P ← CMR[#154]<19:0>;
               INTVEC<4> ← 0 NEXT
               BAILOUT INT
               END
          )
          END;     !END INT
```

```
!CONDITION CODE TESTING ROUTINES

!       THESE ROUTINES TEST CONDITION CODES DURING INSTRUCTION EXECUTION.
!       THE FOLLOWING ROUTINE SETS THE ZERO AND GREATER THAN OR EQUAL
!       ZERO REPEAT CONDITION CODE BITS (RCC<2:1>) FOR SINGLE WORD OPERATIONS.

                CCZG:=  BEGIN
                        RCC<2:1> ← NOT TA0<31> NEXT
                        (IF (TA0 EQL 0) OR (TA0 EQL ONE32) => RCC<2:1> ← #3)
                        END;

!       THE FOLLOWING ROUTINE SETS THE ZERO AND GREATER THAN OR EQUAL
!       ZERO REPEAT CONDITION CODE BITS (RCC<2:1>) FOR DOUBLE WORD OPERATIONS.

                CCZGD:= BEGIN
                        RCC<2:1> ← NOT TD0<63> NEXT
                        (IF (TD0 EQL 0) OR (TD0 EQL (NOT 0<63:0>)) =>
                                RCC<2:1> ← #3)
                        END;

!       THE FOLLOWING ROUTINES SET THE OVERFLOW CONDITION CODE
!       BIT IN ADDITION TO THE ZERO AND GREATER THAN OR EQUAL TO ZERO
!       BITS (RCC<2:1>).  SIGN BIT CORRECTED DURING OVERFLOW.

!       SINGLE WORD OPERATIONS:

                CCOZG:= BEGIN
                        CC<3> ← 0; RCC<2> ← 0 NEXT
                        (IF (TA0<31> EQV TA1<31>) =>
                            CC<3> ← (TA0<31> NEQ TAC<31>) NEXT
                            TAC<31> ← TAC<31> XOR CC<3>
                        ) NEXT
                        RCC<1> ← NOT TAC<31> NEXT
                        (IF (TAC<31:0> EQL 0) OR (TAC<31:0> EQL ONE32) =>
                                RCC<2:1> ← #3)
                        END;

!       DOUBLE WORD OPERATIONS:

                CCOZGD:=BEGIN
                        CC<3> ← 0; RCC<2> ← 0 NEXT
                        (IF TD0<63> EQV TD1<63> =>
                            CC<3> ← (TD0<63> NEQ TDAC<63>) NEXT
                            TDAC<63> ← TDAC<63> XOR CC<3>
                        ) NEXT
                        RCC<1> ← NOT TDAC<63> NEXT
                        (IF (TDAC<63:0> EQL 0) OR (TDAC<63:0> EQL (NOT 0<63:0>)) =>
                                RCC<2:1> ← #3)
                        END;
```

```
! FORMAT I INSTRUCTION EXECUTION
! NOTE: ARITHMETIC IS 32 BIT ONE'S COMPLEMENT WITH MOST
!              SIGNIFICANT BIT THE SIGN

        LA:=    ! LOAD A
                BEGIN
                RDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                TA0 ← TA2;
                A0 ← A NEXT
                PUTA0 NEXT                  ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;     !END LA


        LXB:=   ! LOAD A AND INDEX B
                BEGIN
                LA NEXT                     ! USE "LA" INSTRUCTION
                B0 ← B NEXT
                GETB NEXT                   ! INDEX REGISTER RETURNS IN "TB"
                TB1 ← TBD + 1 NEXT
                TBD ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                PUTB                        ! STORE (TB) IN INDEX REGISTER (B0)
                END;     !END LXB


        LDIF:=  ! LOAD DIFFERENCE
                BEGIN
                RDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                GETA NEXT                   ! (AC(A)) RETURNS IN "TA0"
                TA0 ← NOT TA0 NEXT
                TAC ← TA2 + TA0 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                TA1 ← TA2 NEXT
                CC02G NEXT
                TA1 ← TAC<31:0> NEXT
                PUTA1                       ! STORE (TA1) IN AC(A0+1)
                END;     !END LDIF


        ANA:=   ! SUBTRACT A
                BEGIN
                RDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                GETA NEXT                   ! (AC(A)) RETURNS IN "TA0"
                TA2 ← NOT TA2 NEXT
                TA1 ← TA2;
                TAC ← TA0 + TA2 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CC02G NEXT
                TA0 ← TAC<31:0> NEXT
                PUTA0                       ! STORE (TA0) IN ACCUMULATOR (A0)
                END;     !END ANA
```

```
! FORMAT I INSTRUCTION EXECUTION (PAGE 2)

        AA:=    ! ADD A
                BEGIN
                RDFMTI NEXT              ! OPERAND RETURNS IN "TA2"
                GETA NEXT                ! (AC(A)) RETURNS IN "TA0"
                TA1 ← TA2;
                TAC ← TA0 + TA2 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CC0ZG NEXT
                TA0 ← TAC<31:0> NEXT
                PUTA0                    ! STORE (TA0) IN ACCUMULATOR (A0)
                END;    !END AA

        LSUM:=  ! LOAD SUM
                BEGIN
                RDFMTI NEXT              ! OPERAND RETURNS IN "TA2"
                GETA NEXT                ! (AC(A)) RETURNS IN "TA0"
                TAC ← TA0 + TA2 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                TA1 ← TA2 NEXT
                CC0ZG NEXT
                TA1 ← TAC<31:0> NEXT
                PUTA1                    ! STORE (TA1) IN AC(A0+1)
                END;    !END LSUM

        LNA:=   ! LOAD NEGATIVE
                BEGIN
                RDFMTI NEXT              ! OPERAND RETURNS IN "TA2"
                A0 ← A;
                TA0 ← NOT TA2 NEXT
                PUTA0 NEXT               ! STORE (TA0) IN ACCUMULATOR (A0)
                CCZG
                END;    !END LNA

        LM:=    ! LOAD MAGNITUDE
                BEGIN
                RDFMTI NEXT              ! OPERAND RETURNS IN "TA2"
                A0 ← A;
                TA0 ← TA2 NEXT
                (IF TA0<31> => TA0 ← NOT TA0) NEXT
                PUTA0 NEXT               ! STORE (TA0) IN ACCUMULATOR (A0)
                CCZG
                END;    !END LM

        LB:=    ! LOAD B
                BEGIN
                RDFMTI NEXT              ! OPERAND RETURNS IN "TA2"
                B0 ← A NEXT
                GETB NEXT                ! INDEX REGISTER RETURNS IN "TB"
                (IF A NEQ 0 => TB0 ← TA2<15:0>) NEXT
                PUTB                     ! STORE (TB) IN INDEX REGISTER (B0)
                END;    !END LB
```

```
! FORMAT I INSTRUCTION EXECUTION (PAGE 3)

        AB:=    ! ADD B
                BEGIN
                RDFMTI NEXT               ! OPERAND RETURNS IN "TAZ"
                B0 ← A NEXT
                GETB NEXT                 ! INDEX REGISTER RETURNS IN "TB"
           •    (IF A NEQ 0 =>
                    TAC ← TBD + TAZ NEXT !TBD ZERO EXTENDED
                    TBD ← (TAC<31:0> + TAC<32>)<15:0>
                ) NEXT
                PUTB                      ! STORE (TB) IN INDEX REGISTER (B0)
                END;    !END AB


        ANB:=   ! SUBTRACT B
                BEGIN
                RDFMTI NEXT               ! OPERAND RETURNS IN "TAZ"
                B0 ← A NEXT
                GETB NEXT                 ! INDEX REGISTER RETURNS IN "TB"
                (IF A NEQ 0 =>
                    TAC ← TBD + (NOT TAZ) NEXT
                    TBD ← (TAC<31:0> + TAC<32>)<15:0>
                ) NEXT
                PUTB                      ! STORE (TB) IN INDEX REGISTER (B0)
                END;    !END ANB


        SB:=    ! STORE B
                BEGIN
                B0 ← A NEXT
                GETB NEXT                 ! INDEX REGISTER RETURNS IN "T
                TAZ ← TBD NEXT
                WTFMTI
                END;    !END SB


        SA:=    ! STORE A
                BEGIN
                GETA NEXT                 ! (AC(A)) RETURNS IN "TA0"
                TAZ ← TA0 NEXT
                WTFMTI NEXT               ! REPLACE OPERAND FROM "TAZ"
                CCZG
                END;    !END SA
```

```
! FORMAT I INSTRUCTION EXECUTION (PAGE 4)

        SXB:=   ! STORE A AND INDEX B
                BEGIN
                SA NEXT            != USE STORE A INSTRUCTION
                B0 ← B NEXT
                GETB NEXT                    ! INDEX REGISTER RETURNS IN "TB"
                TB1 ← TBD + 1 NEXT
                TBD ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                PUTB                         ! STORE (TB) IN INDEX REGISTER (B0)
                END;      !END SXB


        SNA:=   ! STORE NEGATIVE
                BEGIN
                GETA NEXT                    ! (AC(A)) RETURNS IN "TA0"
                TA0 ← (NOT TA0) NEXT
                TA2 ← TA0 NEXT
                WTFMTI NEXT                  ! REPLACE OPERAND FROM "TA2"
                CC2G
                END;      !END SNA


        SM:=    ! STORE MAGNITUDE
                BEGIN
                GETA NEXT                    ! (AC(A)) RETURNS IN "TA0"
                (IF TA0<31> => TA0 ← (NOT TA0)) NEXT
                TA2 ← TA0 NEXT
                WTFMTI NEXT                  ! REPLACE OPERAND FROM "TA2"
                CC2G
                END;      !END SM


        BZ:=    ! CLEAR BIT
                BEGIN
                READOP;
                MASK ← 1 NEXT
                MASK ← MASK ^SL0 AK NEXT
                MASK ← NOT MASK NEXT
                MDBR ← MDBR AND MASK NEXT
                MWO NEXT
                TA0 ← MDBR NEXT
                CC2G
                END;      !END BZ
```

! FORMAT I INSTRUCTION EXECUTION (PAGE 5)

```
        BS:=    ! SET BIT
                BEGIN
                READOP;
                MASK ← 1 NEXT
                MASK ← MASK SL0 AK NEXT          !* ACTION TAKEN WHEN AK>#37 ?
                MOBR ← MOBR OR MASK NEXT
                MWO NEXT
                TA0 ← MOBR NEXT
                CCZG
                END;     !END BS


        RA:=    ! REPLACE ADD
                BEGIN
                RDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                GETA NEXT                   ! (AC(A)) RETURNS IN "TA0"
                TA1 ← TA2;
                TAC ← TA2 + TA0 NEXT
                TAC ← TAC<31:0> + TAC <32> NEXT
                CCOZG NEXT
                TA2 ← TAC<31:0> NEXT
                TA1 ← TA2 NEXT
                PUTBACK NEXT
                RPFMTI;                     ! REPLACE OPERAND FROM "TA2"
                PUTA1                       ! STORE (TA1) IN AC(A0+1)
                END;     !END RA


        RI:=    ! REPLACE INCREMENT
                BEGIN
                RDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                TA0 ← 1;
                TA1 ← TA2;
                TAC ← TA2 + 1 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CCOZG NEXT
                TA2 ← TAC<31:0>;
                TA0 ← TAC<31:0>;
                A0 ← A NEXT
                PUTBACK NEXT
                RPFMTI;                     ! REPLACE OPERAND FROM "TA2"
                PUTA0                       ! STORE (TA0) IN ACCUMULATOR (A0)
                END;     !END RI
```

! FORMAT I INSTRUCTION EXECUTION (PAGE 6)

```
        RAN:=   ! REPLACE SUBTRACT
                BEGIN
                PDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                GETA NEXT                   ! (AC(A)) RETURNS IN "TA0"
                TA0 ← NOT TA0;
                TA1 ← TA2;
                TAC ← TA2 + TA0 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CCOZG NEXT
                TA2 ← TAC<31:0> NEXT
                TA1 ← TA2 NEXT
                PUTBACK NEXT
                RPFMTI;                     ! REPLACE OPERAND FROM "TA2"
                PUTA1                       ! STORE (TA1) IN AC(A0+1)
                END;     !END RAN

        RD:=    ! REPLACE DECREMENT
                BEGIN
                PDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                TA0 ← (NOT 1<31:0>);
                TA1 ← TA2;
                TAC ← TA2 + TA0 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CCOZG NEXT
                TA2 ← TAC<31:0>;
                TA0 ← TAC<31:0>;
                A0 ← A NEXT
                PUTBACK NEXT
                RPFMTI;                     ! REPLACE OPERAND FROM "TA2"
                PUTA0                       ! STORE (TA0) IN ACCUMULATOR (A0)
                END;     !END RD
```

```
! FORMAT 1 INSTRUCTION EXECUTION (PAGE 7)

        M.:=    ! MULTIPLY A
                BEGIN
                PDFMTJ NEXT                 ! OPERAND RETURNS IN "TA2"
                GETA NEXT                   ! (AC<A>) RETURNS IN "TA0"
                SIGN ← (TA2<31> XOR TA0<31>) NEXT
                (IF TA2<31> => TA2 ← (NOT TA2));
                (IF TA0<31> => TA0 ← (NOT TA0)) NEXT
                TD0 ← TA2 * TA0 NEXT
                (IF SIGN => TD0 ← (NOT TD0)) NEXT
                TA1 ← TD0<63:32>;
                TA0 ← TD0<31:0> NEXT
                PUTA1;                      ! STORE (TA1) IN AC(A0+1)
                PUTA0 NEXT                  ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2GD
                END;    !END M


        D.:=    ! DIVIDE A
                BEGIN
                PDFMTJ NEXT                 ! OPERAND RETURNS IN "TA2"
                (IF (TA2 EQL 0) OR (TA2 EQL ONE32) =>    ! STOP A ZERO DIVIDE
                    BEGIN
                    CC<3> ← 1 NEXT
                    BAILOUT ICYCLE
                    END
                ) NEXT
                GETD NEXT
                SIGN ← (TA1<31> XOR TA2<31>);
                SIGN1 ← (TA1<31>) NEXT
                (IF SIGN1 => TD0 ← (NOT TD0));
                (IF TA2<31> => TA2 ← (NOT TA2)) NEXT
                TA0 ← (TD0 / TA2)<31:0> NEXT
                TA1 ← (TD0 MINUS (TA0 * TA2))<31:0> NEXT
                (IF SIGN => TA0 ← NOT TA0);
                (IF SIGN1 => TA1 ← NOT TA1) NEXT
                PUTA1;                      ! STORE (TA1) IN AC(A0+1)
                PUTA0;                      ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G;
                CC<3> ← (TD0<63:31> NEQ 0)
                END;    !END D
```

```
! FORMAT I INSTRUCTION EXECUTION (PAGE 8)

        BC:=    ! COMPARE BIT TO ZERO
                BEGIN
                COMPAR ← 1;
                READOP;
                MASK ← 1 NEXT
                MASK ← MASK ISL0 AK NEXT
                MOBR ← MOBR AND MASK NEXT
                CC<2:1> ← 0 NEXT
                (IF (MOBR EQL 0) => CC<2> ← 1)
                END;     !END BC


        CXI:=   ! COMPARE INDEX INCREMENT
                BEGIN
                COMPAR ← 1;
                RDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                B0 ← A NEXT
                GETB NEXT                   ! INDEX REGISTER RETURNS IN "TB"
                CC<0> ← 0 NEXT
                (DECODE ((TBD GTR TA2<15:0>) XOR (TBD<15> NEQ TA2<15>))
                        OR (TBD EQL TA2<15:0>)   =>
                \0      (TB1 ← TBD + 1 NEXT
                        TBD ← (TB1<15:0> + TB1<16>)<15:0> );
                \1      BEGIN
                        CC<0> ← 1;
                        TBD ← 0
                        END
                ) NEXT
                PUTB                        ! STORE (TB) IN INDEX REGISTER (B0)
                END;     !END CXI


        C.:=    ! COMPARE
                BEGIN
                COMPAR ← 1;
                RDFMTI NEXT                 ! OPERAND RETURNS IN "TA2"
                GETA;                       ! (AC(A)) RETURNS IN "TA0"
                CC<2:1> ← 0 NEXT
                (IF TA0 EQL ONE32 => TA0 ← 0);
                (IF TA2 EQL ONE32 => TA2 ← 0) NEXT
                TSTR ← (TA0 TST TA2) NEXT
                (IF (TA0<31> XOR TA2<31>) => TSTR ← (2 - TSTR)<1:0>) NEXT
                (DECODE TSTR =>
                \0      NO.OP;
                \1      CC<2:1> ← 3;
                \2      CC<1> ← 1;
                \3      NO.OP
                )
                END;     !END C
```

```
! FORMAT I INSTRUCTION EXECUTION (PAGE 9)

        CL:=    ! COMPARE LIMITS
                BEGIN
                COMPAR ← 1;
                CC<0> ← 0;
                GETD;
                RDFMT1 NEXT                ! OPERAND RETURNS IN "TA2"
                (IF TA0 EQL ONE32 => TA0 ← 0);
                (IF TA1 EQL ONE32 => TA1 ← 0);
                (IF TA2 EQL ONE32 => TA2 ← 0) NEXT
                (IF ((TA0 GTR TA2) OR (TA2 GEQ TA1)) => CC<0> ← 1)
                END;     !END CL

        CM:=    ! COMPARE MASKED
                BEGIN
                COMPAR ← 1;
                CC<2:1> ← 0;
                GETD;
                RDFMT1 NEXT                ! OPERAND RETURNS IN "TA2"
                (IF TA1 EQL ONE32 => TA1 ← 0);
                TA2 ← TA2 AND TA0 NEXT
                (IF TA2 EQL ONE32 => TA2 ← 0) NEXT
                TSTR ← (TA1 TST TA2) NEXT
                (IF (TA1<31> XOR TA2<31>) => TSTR ← (2 - TSTR)<1:0>) NEXT
                (DECODE TSTR =>
                \0      NO.OP;
                \1      CC<2:1> ← 3;
                \2      CC<1> ← 1;
                \4      NO.OP
                )
                END;     !END CM

        CG:=    ! COMPARE GATED
                BEGIN
                COMPAR ← 1;
                CC<2:1> ← 0;
                GETD;
                RDFMT1 NEXT                ! OPERAND RETURNS IN "TA2"
                TAC ← (TA2 + (NOT TA0)) NEXT
                TA2 ← (TAC<31:0> + TAC<32>)<31:0> NEXT
                (IF TA2<31> => TA2 ← (NOT TA2)) NEXT
                (IF TA1 EQL ONE32 => TA1 ← 0) NEXT
                TSTR ← (TA2 TST TA1) NEXT
                (IF TA1<31> => TSTR ← (2 - TSTR)<1:0>) NEXT
                (DECODE TSTR =>
                \0      NO.OP;
                \1      CC<2:1> ← 3;
                \2      CC<1> ← 1;
                \3      NO.OP
                )
                END;     !END CG
```

```
! FORMAT I INSTRUCTION EXECUTION (PAGE 10)

        LCT:=   ! LOAD CMP TASK
                BEGIN
                (IF PPFLAG => ILOCK ← 1);
                READOP NEXT
                (IF A GEQ #6 => K ← 0) NEXT
                (DECODE A =>
                \0      CMR[AK] ← MOBR;
                \1      CMR[AK] ← MOBR<19:0>;
                \2      (CKPRIV NEXT CMR[AK] ← MOBR<17:0>);
                \3      NO.OP;
                \4      NO.OP;
                \5      NO.OP;
                \6      (CKPRIV NEXT CMR[AK] ← MOBR<19:0>);
                \7      (CKPRIV NEXT CMR[AK] ← MOBR<22:0>)
                ) NEXT
                (IF PPFLAG => AK ← (AK + 1)<5:0>)
                END;    !END LCT


        LCI:=   ! LOAD CMP INTERRUPT
                BEGIN
                (IF PPFLAG => ILOCK ← 1);
                CKPRIV NEXT
                READOP NEXT
                (DECODE A =>
                \0      CMR[(AK + #100)<6:0>] ← MOBR;
                \1      CMR[(AK + #100)<6:0>] ← MOBR<19:0>;
                \2      CMR[(AK + #100)<6:0>] ← MOBR<17:0>;
                \3      NO.OP;
                \4      CMR[(AK + #100)<6:0>] ← MOBR<19:0>;
                \5      CMR[(AK + #100)<6:0>] ← MOBR<19:0>;
                \6      CMR[(AK + #100)<6:0>] ← MOBR<20:0>;
                \7      CMR[(AK + #100)<6:0>] ← MOBR<20:0>
                ) NEXT
                (IF RPFLAG => AK ← (AK + 1)<5:0>)
                END;    !END LCI
```

```
! FORMAT 1 INSTRUCTION EXECUTION (PAGE 11)

        SCT:=   ! STORE CMR TASK
                BEGIN
                (IF RPFLAG => ILOCK ← 1);
                (IF A GEQ #6 => K ← 0) NEXT
                (DECODE A =>
                \0      MOBR ← CMR[AK];
                \1      MOBR ← CMR[AK];
                \2      (CKPRIV NEXT MOBR ← CMR[AK]);
                \3      NO.OP;
                \4      NO.OP;
                \5      NO.OP;
                \6      (CKPRIV NEXT MOBR ← CMR[AK]);
                \7      (CKPRIV NEXT MOBR ← CMR[AK])
                ) NEXT
                (IF (A LSS #3) OR (A GTR #5) =>
                    BEGIN
                    OPADI NEXT
                    MWD
                    END
                ) NEXT
                (IF RPFLAG => AK ← (AK + 1)<5:0>)
                END;    !END SCT


        SCI:=   ! STORE CMR INTERRUPT
                BEGIN
                (IF RPFLAG => ILOCK ← 1);
                CKPRIV NEXT
                OPADI NEXT
                (IF A NEQ #3 =>
                    MOBR ← CMR[(AK + #100)<6:0>] NEXT
                    MWD
                ) NEXT
                (IF RPFLAG => AK ← (AK + 1)<5:0>)
                END;    !END SCI
```

```
! FORMAT I INSTRUCTION DECODE TABLE

        FMTI:=  BEGIN
                (IF F0 EQL 1 =>
                    (DECODE F1 =>
                        LA;               ! 10    LOAD A
                        LXB;              ! 11    LOAD A AND INDEX B
                        LDIF;             ! 12    LOAD DIFFERENCE
                        ANA;              ! 13    SUBTRACT A
                        AA;               ! 14    ADD A
                        LSUM;             ! 15    LOAD SUM
                        LNA;              ! 16    LOAD NEGATIVE
                        LM                ! 17    LOAD MAGNITUDE
                    )
                ));
                (IF F0 EQL 2 =>
                    (DECODE F1 =>
                        LB;               ! 20    LOAD B
                        AB;               ! 21    ADD B
                        ANB;              ! 22    SUBTRACT B
                        SB;               ! 23    STORE B
                        SA;               ! 24    STORE A
                        SXB;              ! 25    STORE A AND INDEX B
                        SNA;              ! 26    STORE NEGATIVE
                        SM                ! 27    STORE MAGNITUDE
                    )
                ));

                (IF F0 EQL 3 =>
                    (DECODE F1 =>
                        OPEX;             ! 30
                        OPEX;             ! 31
                        BZ;               ! 32    CLEAR BIT
                        BS;               ! 33    SET BIT
                        RA;               ! 34    REPLACE ADD
                        RI;               ! 35    REPLACE INCREMENT
                        RAN;              ! 36    REPLACE SUBTRACT
                        RD                ! 37    REPLACE DECREMENT
                    )
                ));
                (IF F0 EQL 4 =>
                    (DECODE F1 =>
                        M.;               ! 40    MULTIPLY A
                        D.;               ! 41    DIVIDE A
                        DC;               ! 42    COMPARE BIT TO ZERO
                        CXI;              ! 43    COMPARE INDEX INCREMENT
                        C.;               ! 44    COMPARE
                        CL;               ! 45    COMPARE LIMITS
                        CM;               ! 46    COMPARE MASKED
                        CG                ! 47    COMPARE GATED
                    )
                ));
```

```
! FORMAT I INSTRUCTION DECODE TABLE (continued)

            (IF F EQL #54 => LCT);
                                        ! 54    LOAD CMR TASK
            (IF F EQL #55 => LCT);
                                        ! 55    LOAD CMR INTERRUPT
            (IF F EQL #56 => SCT);
                                        ! 56    STORE CMR TASK
            (IF F EQL #57 => SCT)
                                        ! 57    STORE CMR INTERRUPT
            END;     !END FORMAT I
```

```
! FORMAT II INSTRUCTION EXECUTION

        OR.:=   ! INCLUSIVE OR
                BEGIN
                GETA;                   ! (AC(A)) RETURNS IN "TA0"
                READOP  NEXT
                TA0 ← TA0 OR MOBR NEXT
                PUTA0 NEXT              ! STORE (TA0) IN ACCUMULATOR (A0)
                CCZG
                END;    ! END OF OR.


        SC:=    ! SELECTIVE CLEAR
                BEGIN
                GETA;                   ! (AC(A)) RETURNS IN "TA0"
                READOP NEXT
                TA0 ← TA0 AND (NOT MOBR) NEXT
                PUTA0 NEXT              ! STORE (TA0) IN ACCUMULATOR (A0)
                CCZG
                END;    !END SC


        MS:=    ! SELECTIVE SUBSTITUTE
                BEGIN
                GETD;
                READOP NEXT
                TA1 ← ((TA0 AND MOBR) OR (TA1 AND (NOT TA0))) NEXT
                PUTA1 NEXT              ! STORE (TA1) IN AC(A0+1)
                TA0 ← TA1 NEXT
                CCZG
                END;    !END MS


        XOR.:=  ! EXCLUSIVE OR
                BEGIN
                GETA;                   ! (AC(A)) RETURNS IN "TA0"
                READOP NEXT
                TA0 ← TA0 XOR MOBR NEXT
                PUTA0 NEXT              ! STORE (TA0) IN ACCUMULATOR (A0)
                CCZG
                END;    !END XOR
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 2)

        ALP:=   ! ADD LOGICAL PRODUCT
                BEGIN
                GETD;
                READOP NEXT
                TA0 ← MOBR AND TA0 NEXT
                TAC ← TA1 + TA0 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CCOZG NEXT
                TA1 ← TAC<31:0> NEXT
                PUTA1                   ! STORE (TA1) IN AC(A0+1)
                END;    !END ALP


        LLP:=   ! LOAD LOGICAL PRODUCT
                BEGIN
                GETA;                   ! (AC(A)) RETURNS IN "TA0"
                READOP NEXT
                TA0 ← TA0 AND MOBR NEXT
                PUTA0 NEXT              ! STORE (TA0) IN ACCUMULATOR (A0)
                CCZG
                END;    !END LLP


        NLP:=   ! SUBTRACT LOGICAL PRODUCT
                BEGIN
                GETD;
                READOP NEXT
                TA0 ← NOT (MOBR AND TA0) NEXT
                TAC ← TA1 + TA0 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CCOZG NEXT
                TA1 ← TAC<31:0> NEXT
                PUTA1                   ! STORE (TA1) IN AC(A0+1)
                END;    !END NLP
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 3)

        LLPN:=  ! LOAD LOGICAL PRODUCT NEXT
                BEGIN
                GETD;
                READOP NEXT
                TA1 ← TA0 AND MOBR NEXT
                PUTA1 NEXT              ! STORE (TA1) IN AC(A0+1)
                TA0 ← TA1 NEXT
                CC2G
                END;    !END LLPN


        CNT:=   ! COUNT ONES
                BEGIN
                GETA;                   ! [illegible] .TURNS IN "TA0"
                READOP NEXT
                TA0 ← 0;
                COUNT ← 32 NEXT
                CNTLP:= BEGIN
                        IF COUNT NEQ 0 =>
                            BEGIN
                            (IF MOBR<0> => TA0 ← (TA0 + 1)<31:0>) NEXT
                            MOBR ← MOBR SRR 1;
                            COUNT ← (COUNT MINUS 1)<5:0> NEXT
                            CNTLP
                            END
                        END NEXT
                PUTA0 NEXT              ! STORE (TA0) IN ACCUMULATOR <A0>
                CC2G
                END;    !END CNT


        XR:=    ! EXECUTE REMOTE
                BEGIN
                READOP NEXT
                U ← MOBR NEXT
                EXRF ← 1
                END;    !END XR


        XRL:=   ! EXECUTE REMOTE LOWER
                BEGIN
                READOP NEXT
                U<31:16> ← MOBR<15:0> NEXT
                EXRF ← 1
                END;    !END XRL
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 4)

        SLP:=   ! STORE LOGICAL PRODUCT
                BEGIN
                GETD;
                READOP NEXT
                MOBR ← TA0 AND TA1 NEXT
                WRITOP NEXT
                TA0 ← MOBR NEXT
                CCZG
                END;    !END SLP


        SSUM:=  ! STORE SUM
                BEGIN
                GETD NEXT
                TAC ← TA0 + TA1 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CCOZG NEXT
                MOBR ← TAC<31:0>;
                TA1 ← TAC<31:0> NEXT
                WRITOP;
                PUTA1                   ! STORE (TA1) IN AC(A0+1)
                END;    !END SSUM


        SDIF:=  ! STORE DIFFERENCE
                BEGIN
                GETD NEXT
                TA0 ← NOT TA0 NEXT
                TAC ← TA1 + TA0 NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CCOZG NEXT
                MOBR ← TAC<31:0>;
                TA1 ← TAC<31:0> NEXT
                WRITOP;
                PUTA1                   ! STORE (TA1) IN AC(A0+1)
                END;    !END SDIF
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 5)

        DS:=     ! DOUBLE STORE A
                 BEGIN
                 GETD NEXT
                 MDBR ← TA0 NEXT
                 WRITOP NEXT
                 MDAR ← (MDAR + 1)<17:0> NEXT
                 MDBR ← TA1 NEXT
                 MWO NEXT
                 CC2GO
                 END;     !END DS


        ROR:=    ! REPLACE INCLUSIVE OR
                 BEGIN
                 OR. NEXT                    != JUST LIKE AN "OR"
                 MDBR ← TA0 NEXT
                 PUTBACK NEXT
                 WRITOP
                 END;     ! END ROR


        RSC:=    ! REPLACE SELECTIVE CLEAR
                 BEGIN
                 SC NEXT                     != JUST AN "SC"
                 MDBR ← TA0 NEXT
                 PUTBACK NEXT
                 WRITOP
                 END;     !END RSC


        RMS:=    ! REPLACE SELECTIVE SUBSTITUTE
                 BEGIN
                 MS NEXT                     != JUST LIKE SELECTIVE SUB
                 MDBR ← TA1 NEXT
                 PUTBACK NEXT
                 WRITOP
                 END;     !END RMS


        RXOR:=   ! REPLACE EXCLUSIVE OR
                 BEGIN
                 XOR. NEXT
                 MDBR ← TA0 NEXT
                 PUTBACK NEXT
                 WRITOP
                 END;     !END RXOR
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 6)

        RALP:=  ! REPLACE A + LOGICAL PRODUCT
                BEGIN
                ALP NEXT                  != JUST LIKE "ALP"
                MOBR ← TA1 NEXT
                PUTBACK NEXT
                WRITOP
                END;    !END RALP


        RLP:=   ! REPLACE LOGICAL PRODUCT
                BEGIN
                LLPN NEXT                 != JUST LIKE "LLPN"
                MOBR ← TA1 NEXT
                PUTBACK NEXT
                WRITOP
                END;    !END RLP


        RNLP:=  ! REPLACE A - LOGICAL PRODUCT
                BEGIN
                NLP NEXT                  != LIKE AN "NLP"
                MOBR ← TA1 NEXT
                PUTBACK NEXT
                WRITOP
                END;    !END RNLP


        TSF:=   !TEST AND SET FLAG
                BEGIN
                READOP NEXT
                (DECODE MOBR<31> =>
                \0      BEGIN
                        MOBR<31> ← 1 NEXT
                        MWO;
                        CC<2> ← 1
                        END;

                \1      CC<2> ← 0
                )
                END;    !END TSF
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 7)

        DL:=    ! DOUBLE LOAD A
                BEGIN
                AA ← A;
                READD NEXT
                TD0 ← TD1 NEXT
                PUTD;
                CC2GD
                END;     !END DL


        DA:=    !DOUBLE ADD A
                BEGIN
                GETD;
                READD NEXT
                TDAC ← TD0 + TD1 NEXT
                TDAC ← TDAC<63:0> + TDAC<64> NEXT
                CC02GD NEXT
                TD0 ← TDAC<63:0> NEXT
                PUTD
                END;     !END DA


        DAN:=   ! DOUBLE SUBTRACT A
                BEGIN
                GETD;
                READD NEXT
                TD1 ← NOT TD1 NEXT
                TDAC ← TD0 + TD1 NEXT
                TDAC ← TDAC<63:0> + TDAC<64> NEXT
                CC02GD NEXT
                TD0 ← TDAC<63:0> NEXT
                PUTD
                END;     !END DAN
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 8)

        DC :=    ! DOUBLE COMPARE
                 BEGIN
                 COMPAR ← 1;
                 GETD;
                 READD NEXT
                 (IF TD0 EQL ONE32@ONE32 => TD0 ← 0);
                 (IF TD1 EQL ONE32@ONE32 => TD1 ← 0) NEXT
                 CC<2:1> ← 0 NEXT
                 TSTR ← (TD0 TST TD1) NEXT
                 (IF (TD1<63> XOR TD0<63>) => TSTR ← (2 - TSTR)<1:0>) NEXT
                 (DECODE TSTR =>
                 \0       NO.OP;
                 \1       CC<2:1> ← 3;
                 \2       CC<1> ← 1;
                 \3       NO.OP
                 )
                 END;     !END DC

        LBMP :=  ! LOAD BASE AND MEMORY PROTECTION
                 BEGIN
                 B0 ← B NEXT
                 GETB NEXT                   ! INDEX REGISTER RETURNS IN "TB"
                 TB1 ← Y + TBD NEXT
                 TDD ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                 (IF TBD<0> =>               ! ODD ADDRESS -- ERROR
                     BEGIN
                     INTVEC<2> ← 1;
                     ISC ← #2 NEXT                     !ILLEGAL INSTRUCTION
                     BAILOUT ICYCLE
                     END
                 ) NEXT
                 (IF ASP<19:18> EQL 0 =>
                     BEGIN
                     IF (NOT (ASR<8> AND (B EQL #7) AND (A NEQ #7))) =>
                         BEGIN
                         INTVEC<2> ← 1;
                         ISC ← #3 NEXT                 !PRIVILEGED INSTRUCTION
                         BAILOUT ICYCLE
                         END
                     END
                 ) NEXT
                 READOP NEXT
                 RST[A]<17:0> ← MDBR<17:0> NEXT
                 MDAR ← (MDAR + 1)<17:0> NEXT
                 MPD NEXT
                 SPR[A]<20:0> ← MDBR<20:0>;
                 SIR[A]<19:17> ← B;
                 SIR[A]<16:0> ← TBD
                 END;     !END LBMP
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 9)

        XSIPI:= ! ENTER EXECUTIVE STATE/INTERPROCESSOR INTERRUPT
                BEGIN
                DECODE (A NEQ 0) =>
                XS:=    BEGIN
                        B0 ← B NEXT
                        GETB NEXT       ! INDEX REGISTER RETURNS IN "TB"
                        TB1 ← SY + TDD NEXT
                        ISC ← (TB1<15:0> + TB1<16>)<15:0>;
                        INTVEC<4> ← 1
                        END;
                IPI:=   CKPRIV)
                END;    !END XSIPI


        AEI:=   ! ALLOW ENABLE INTERRUPT
                BEGIN                   !**NOT IMPLEMENTED**
                CKPRIV
                END;    !END AEI


        PEI:=   ! PREVENT ENABLE INTERRUPT
                BEGIN                   !**NOT IMPLEMENTED**
                CKPRIV
                END;    !END PEI


        LIM:=   !LOAD IOC MONITOR CLOCK
                BEGIN                   !**NOT IMPLEMENTED**
                NO.OP
                END;    !END LIM


        IO:=    ! INITIATE I/O
                BEGIN
                READOP NEXT
                (IF A LEQ 3 => IOC[A<1:0>] ← MOBR)
                END;    !END IO
```

```
! FORMAT II INSTRUCTION EXECUTION (PAGE 10)

        IR:=    ! INTERRUPT RETURN
                BEGIN
                (IF ASR<19> =>                          !CLASS I
                    NDR<0> ← 0;
                    IFLAG ← 0;
                    ASR ← CMR[#141]<19:0>;
                    P ← CMR[#143]<19:0> NEXT
                    BAILOUT ICYCLE) NEXT
                (IF ASR<18> =>                          !CLASS II
                    NDR<1> ← 0;
                    IFLAG ← 0;
                    ASR ← CMR[#145]<19:0>;
                    P ← CMR[#147]<19:0> NEXT
                    BAILOUT ICYCLE) NEXT
                (IF ASR<17> =>                          !CLASS III
                    IFLAG ← 0;
                    ASR ← CMR[#151]<19:0>;
                    P ← CMR[#153]<19:0> NEXT
                    BAILOUT ICYCLE) NEXT
                (IF ASR<16> =>                          !CLASS IV
                    IFLAG ← 0;
                    ASR ← CMR[#155]<19:0>;
                    P ← CMR[#157]<19:0>)
                END;    !END IR


        RP:=    ! REPEAT
                BEGIN
                RPTA ← A;
                RPTB ← B;
                RPTSY ← SY;
                B0 ← #7 NEXT
                GETB NEXT                   ! INDEX REGISTER RETURNS IN "TB"
                (IF (TBD EQL 0) OR (TBD EQL "FFFF) =>
                    BEGIN
                    PD ← (PD + 1)<15:0> NEXT
                    BAILOUT ICYCLE
                    END
                ) NEXT
                RPFLAG ← 1 NEXT
                READIN NEXT
                BAILOUT ICYCLE
                END;    !END RP
```

```
!       FLOATING POINT INSTRUCTIONS (NOT IMPLEMENTED IN THIS ISP)

!       GETS, PUTS, AND READS ARE PERFORM TO GENERATE DATA ACCESS
!       AND SPECIFIC ADDRESSING.  NO FLOATING POINT COMPUTATION
!       IS PERFORMED.

        FA:=    ! FLOATING-POINT ADD
                BEGIN
                GETD;
                READD NEXT
                TAC←TD0<63:32>+TD1<63:32> NEXT
                TD0<63:32> ← (TAC<31:0> + TAC<32>)<31:0> NEXT
                PUTD
                END;    !END FA


        FAN:=   ! FLOATING-POINT SUBTRACT
                BEGIN
                GETD;
                READD NEXT
                TAC ← TD0<63:32> + (NOT TD1<63:32>) NEXT
                TD0<63:32> ← (TAC<31:0> + TAC<32>)<31:0> NEXT
                PUTD
                END;    !END FAN


        FM:=,   ! FLOATING-POINT MULTIPLY
                BEGIN
                GETD;
                READD NEXT
                SIGN ← TD0<63> XOR TD1<63> NEXT
                (IF TD0<63> => TD0<63:32> ← NOT TD0<63:32>) ;
                (IF TD1<63> => TD1<63:32> ← NOT TD1<63:32>) NEXT
                TD0<63:32> ← ((TD0<63:32> * TD1<63:32>) SRA 16)<31:0> NEXT
                (IF SIGN => TD0<63:32> ← NOT TD0<63:32>) NEXT
                PUTD
                END;    !END FM


        FD:=    ! FLOATING-POINT DIVIDE
                BEGIN
                GETD;
                READD NEXT
                (IF (TD1<63:32> EQL 0) OR (TD1<63:32> EQL ONE32) =>
                        BAILOUT FD) NEXT
                SIGN ← TD0<63> XOR TD1<63> NEXT
                (IF TD0<63> => TD0<63:32> ← NOT TD0<63:32>) ;
                (IF TD1<63> => TD1<63:32> ← NOT TD1<63:32>) NEXT
                TD0<63:32> ← ((TD0<63:32> @ 0<15:0>) / TD1<63:32>)<31:0> NEXT
                (IF SIGN => TD0<63:32> ← NOT TD0<63:32>) NEXT
                PUTD
                END;    !END FD
```

```
1       !FLOATING POINT INSTRUCTIONS (NOT IMPLEMENTED IN THIS ISP)

        FAR:=   ! FLOATING POINT ADD WITH ROUND
                BEGIN
                GETD;
                READD NEXT
                TAC←TD0<63:32>+TD1<63:32> NEXT
                TD0<63:32> ← (TAC<31:0> + TAC<32>)<31:0> NEXT
                PUTD
                END;      !END FAR


        FANR:=  ! FLOATING-POINT SUBTRACT WITH ROUND
                BEGIN
                GETD;
                READD NEXT
                TAC ← TD0<63:32> + (NOT TD1<63:32>) NEXT
                TD0<63:32> ← (TAC<31:0> + TAC<32>)<31:0> NEXT
                PUTD
                END;      !END FANR


        FMR:=   ! FLOATING-POINT MULTIPLY WITH ROUND
                BEGIN
                GETD;
                READD NEXT
                SIGN ← TD0<63> XOR TD1<63> NEXT
                (IF TD0<63> => TD0<63:32> ← NOT TD0<63:32>) ;
                (IF TD1<63> => TD1<63:32> ← NOT TD1<63:32>) NEXT
                TD0<63:32> ← ((TD0<63:32> * TD1<63:32>) tSR0 18)<31:0> NEXT
                (IF SIGN => TD0<63:32> ← NOT TD0<63:32>) NEXT
                PUTD
                END;      !END FMR


        FDR:=   ! FLOATING-POINT DIVIDE WITH ROUND
                BEGIN
                GETD;
                READD NEXT
                (IF (TD1<63:32> EQL 0) OR (TD1<63:32> EQL ONE32) =>
                          BAILOUT FDR) NEXT
                SIGN ← TD0<63> XOR TD1<63> NEXT
                (IF TD0<63> => TD0<63:32> ← NOT TD0<63:32>) ;
                (IF TD1<63> => TD1<63:32> ← NOT TD1<63:32>) NEXT
                TD0<63:32> ← ((TD0<63:32> @ 0<15:0>) / TD1<63:32>)<31:0> NEXT
                (IF SIGN => TD0<63:32> ← NOT TD0<63:32>) NEXT
                PUTD
                END;      !END FDR
```

```
UYH2:ISPEY?IOCAOH]@CMU 100       23-Feb-78 11:06       WO15@CMU-10D       Page 5A-1

! FORMAT 11 INSTRUCTION DECODE TABLE

        FMT11:= BEGIN
                DECODE F1@F2 =>
                      OPEX;                   ! 00 0
                      OPEX;                   ! 00 1
                      OPEX;                   ! 00 2
                      OPEX;                   ! 00 3
                      OPEX;                   ! 00 4
                      OPEX;                   ! 00 5
                      OPEX;                   ! 00 6
                      OPEX;                   ! 00 7
                      OR.;                    ! 01 0  OR
                      SC;                     ! 01 1  SELECTIVE CLEAR A
                      MS;                     ! 01 2  SELECTIVE SUBSTITUTE
                      XOR.;                   ! 01 3  EXCLUSIVE OR
                      ALP;                    ! 01 4  ADD LOGICAL PRODUCT
                      LLP;                    ! 01 5  LOAD LOGICAL PRODUCT
                      NLP;                    ! 01 6  SUBTRACT LOGICAL PRODUCT
                      LLPN;                   ! 01 7  LOAD LOGICAL PRODUCT NEG
                      CNT;                    ! 02 0  COUNT ONES
                      OPEX;                   ! 02 1
                      XP;                     ! 02 2  EXECUTE REMOTE
                      XPL;                    ! 02 3  EXECUTE REMOTE LOWER
                      SLP;                    ! 02 4  STORE LOGICAL PRODUCT
                      SSUM;                   ! 02 5  STORE SUM
                      SDIF;                   ! 02 6  STORE DIFFERENCE
                      DS;                     ! 02 7  DOUBLE STORE A
                      ROR;                    ! 03 0  REPLACE INCLUSIVE OR
                      RSC;                    ! 03 1  REPLACE SELECTIVE CLEAR
                      RMS;                    ! 03 2  REPLACE SELECTIVE SUBSTITUTE
                      RXOR;                   ! 03 3  REPLACE EXCLUSIVE OR
                      RALP;                   ! 03 4  REPLACE A+LOGICAL PRODUCT
                      RLP;                    ! 03 5  REPLACE LOGICAL PRODUCT
                      RNLP;                   ! 03 6  REPLACE A-LOGICAL PRODUCT
                      TSF;                    ! 03 7  TEST AND SET FLAG
                      OPEX;                   ! 04 0
                      OPEX;                   ! 04 1
                      OPEX;                   ! 04 2
                      OPEX;                   ! 04 3
                      OPEX;                   ! 04 4
                      OPEX;                   ! 04 5
                      OPEX;                   ! 04 6
                      OPEX;                   ! 04 7
                      DL;                     ! 05 0  DOUBLE LOAD A
                      DA;                     ! 05 1  DOUBLE ADD A
                      DAN;                    ! 05 2  DOUBLE SUBTRACT A
                      DC;                     ! 05 3  DOUBLE COMPARE
                      LBMP;                   ! 05 4  LOAD BASE AND
                                              !       MEMORY PROTECTION
! FORMAT II INSTRUCTION DECODE TABLE (CONTINUED)

             OPEX;          ! 05 5
             OPEX;          ! 05 6
             OPEX;          ! 05 7
             FA;            ! 06 0  FLOATING-POINT ADD
             FAN;           ! 06 1  FLOATING-POINT MULTIPLY
            'FM;            ! 06 2  FLOATING-POINT SUBTRACT
             FD;            ! 06 3  FLOATING-POINT DIVIDE
             FAR;           ! 06 4  FLOATING-POINT ADD
                            !       WITH POUND
             FANR;          ! 06 5  FLOATING-POINT SUBTRACH
                            !       WITH POUND
             FMR;           ! 06 6  FLOATING-POINT MULTIPLY
                            !       WITH POUND
             FDR;           ! 06 7  FLOATING-POINT DIVIDE
                            !       WITH POUND
             XSIPI;         ! 07 0  ENTER EXECUTIVE STATE;
                            !       INTERPROCESSOR INTERRUPT
             AEI;           ! 07 1  ALLOW ENABLE INTERRUPT
             PEI;           ! 07 2  PREVENT ENABLE INTERRUPT
             LTM;.          ! 07 3  LOAD IOC MONITOR CLOCK
             IO;            ! 07 4  INITIATE I/O
             IR;            ! 07 5  INTERRUPT RETURN
             RP;            ! 07 6  REPEAT
             OPEX           ! 07 7
          END;     !END FORMAT II
```

```
! FORMAT III INSTRUCTION EXECUTION

        JEP:=   ! JUMP ON EVEN PARITY
                BEGIN
                OPAD1;
                GETD NEXT
                TA0 ← TA0 AND TA1 NEXT
                PARITY NEXT
                (IF NOT T0 => P ← S0 @ TB1)
                END;    !END JEP


        JOP:=   ! JUMP ON ODD PARITY
                BEGIN
                OPAD1;
                GETD NEXT
                TA0 ← TA0 AND TA1 NEXT
                PARITY NEXT                     ! CHECK PARITY
                (IF T0 => P ← S0 @ TB1)
                END;    !END JOP


        DJZ:=   ! JUMP ON DOUBLE PRECISSION ZERO
                BEGIN
                GETD;
                OPAD1 NEXT
                (IF (TD0 EQL 0) OR (TD0 EQL ONE32@ONE32) => P ← S0 @ TB1)
                END;    !END DJZ


        DJNZ:=  ! JUMP ON DOUBLE PRECISION NOT ZERO
                BEGIN
                GETD;
                OPAD1 NEXT
                (IF (TD0 NEQ 0) AND (TD0 NEQ ONE32@ONE32) => P ← S0 @ TB1)
                END;    !END DJNZ
```

```
! FORMAT III INSTRUCTION EXECUTION (PAGE 2)

        JP:=    ! JUMP ON A POSITIVE
                BEGIN
                GETA;                     ! (AC(A)) RETURNS IN "TA0"
                OPAD1 NEXT
                (IF NOT TA0<31> OR (TA0<31:0> EQL "FFFF) => P ← 50 @ TB1)
                END;    !END JP


        JN:=    ! JUMP ON A NEGATIVE
                BEGIN
                GETA;                     ! (AC(A)) RETURNS IN "TA0"
                OPAD1 NEXT
                (IF TA0<31> AND (TA0<31:0> NEQ "FFFF) => P ← 50 @ TB1)
                END;    !END JN


        JZ:=    ! JUMP ON A ZERO
                BEGIN
                GETA;                     ! (AC(A)) RETURNS IN "TA0"
                OPAD1 NEXT
                (IF (TA0 EQL 0) OR (TA0 EQL ONE32) => P ← 50 @ TB1)
                END;    !END JZ


        JNZ:=   ! JUMP ON A NOT ZERO
                BEGIN
                GETA;                     ! (AC(A)) RETURNS IN "TA0"
                OPAD1 NEXT
                (IF (TA0 NEQ 0) AND (TA0 NEQ ONE32) => P ← 50 @ TB1)
                END;    !END JNZ


        LBJ:=   ! LOAD B AND JUMP
                BEGIN
                B0 ← A;
                TB ← P NEXT
                PUTB;                     ! STORE (TB) IN INDEX REGISTER (B0)
                OPAD1 NEXT
                P ← 50 @ TB1
                END;    !END LBJ
```

```
! FORMAT III INSTRUCTION EXECUTION (PAGE 3)

        JBNZ:=  ! INDEX JUMP B
                BEGIN
                B0 ← A NEXT
                GETB NEXT                    ! INDEX REGISTER RETURNS IN "TB"
                (IF (TBD NEQ 0) AND (TBD NEQ #FFFF) =>
                    BEGIN
                    TB1 ← TBD - 1 NEXT
                    TBD ← (TB1<15:0> + TB1<16>)<15:0> NEXT
                    PUTB NEXT
                    OPAD1 NEXT
                    P ← S0 @ TB1
                    END
                )
                END;     !END JBNZ

        JS:=    ! JUMP SY+B
                BEGIN
                B0 ← B NEXT
                GETB NEXT                    ! INDEX REGISTER RETURNS IN "TB"
                TB1 ← SY + TBD NEXT
                PD ← (TB1<15:0> + TB1<16>)<15:0>;
                PS ← TBS
                END;     !END JS

        JL:=    ! UNCONDITIONAL JUMP LOWER
                BEGIN
                OPAD1 NEXT
                P ← S0 @ TB1 NEXT
                READIN NEXT
                UPLOW ← 1;
                EXPF ← 1
                END;     !END JL

        JNF:=   ! JUMP ON NO OVERFLOW
                BEGIN
                DECODE CC<3> =>
                \0  BEGIN
                    OPAD1 NEXT
                    P ← S0 @ TB1
                    END;
                \1  CC<3> ← 0
                END;     !END JNF

        JOF:=   ! JUMP ON OVERFLOW
                BEGIN
                DECODE CC<3> =>
                \0  NO.OP;
                \1  BEGIN
                    OPAD1 NEXT
                    CC<3> ← 0;
                    P ← S0 @ TB1
                    END
                END;     !END JOF
```

```
! FORMAT III INSTRUCTION EXECUTION (PAGE 4)

        JNE:=   ! JUMP ON NOT EQUAL
                BEGIN
                DECODE CC<2> =>
                \0  BEGIN
                    OPAD1 NEXT
                    P ← S0 @ TB1
                    END;
                \1  NO.OP
                END;    !END JNE


        JE:=    ! JUMP ON EQUAL
                BEGIN
                DECODE CC<2> =>
                \0  NO.OP;
                \1  BEGIN
                    OPAD1 NEXT
                    P ← S0 @ TB1
                    END
                END;    !END JE


        JG:=    ! JUMP ON GREATER THAN
                BEGIN
                IF ((NOT CC<2>) AND CC<1>) =>
                    BEGIN
                    OPAD1 NEXT
                    P ← S0 @ TB1
                    END
                END;    !END JG


        JGE:=   ! JUMP ON GREATER THAN OR EQUAL
                BEGIN
                IF CC<1> =>
                    BEGIN
                    OPAD1 NEXT
                    P ← S0 @ TB1
                    END
                END;    !END JGE


        JLT:=   ! JUMP ON LESS THAN
                BEGIN
                IF CC<2:1> EQL 0 =>
                    BEGIN
                    OPAD1 NEXT
                    P ← S0 @ TB1
                    END
                END;    !END JLT
```

```
! FORMAT III INSTRUCTION EXECUTION (PAGE 5)

        JLE:=   ! JUMP ON LESS THAN OR EQUAL
                BEGIN
                IF CC<2:1> NEQ 1 =>
                    BEGIN
                    OPAD; NEXT
                    P ← SB @ TB;
                    END
                END;    !END JLE


        JNW:=   ! JUMP OUTSIDE LIMITS
                BEGIN
                IF CC<0> =>
                    BEGIN
                    OPAD; NEXT
                    P ← SB @ TB;
                    END
                END;    !END JNW


        JW:=    ! JUMP WITHIN LIMITS
                BEGIN
                IF NOT CC<0> =>
                    BEGIN
                    OPAD; NEXT
                    P ← SB @ TB;
                    END
                END;    !END JW


        RJ:=    ! RETURN JUMP
                BEGIN
                OPAD; NEXT
                MDBR ← P NEXT
                MWD NEXT
                P ← SB @ TB; NEXT
                PD ← (PD + 1)<15:0>
                END;    !END RJ


        RJC:=   ! RETURN JUMP
                BEGIN
                IF A EQL JUMPSW => RJ
                END;    !END RJC
```

```
! FORMAT III INSTRUCTION EXECUTION (PAGE 6)

        RJSC:=  ! RETURN JUMP
                BEGIN
                CKPRIV NEXT
                RJ NEXT
                (IF A EQL 4 => STOP);
                (IF A EQL STOPSW => STOP)
                END;    !END RJSC


        J:=     ! MANUAL JUMP
                BEGIN
                OPAD1 NEXT
                P ← S0 @ TB1
                END;    !END J


        JC:=    ! MANUAL JUMP
                BEGIN
                IF A EQL JUMPSW => J
                END;    !END JC


        JSC:=   ! MANUAL JUMP
                BEGIN
                CKPRIV NEXT
                J NEXT
                (IF A EQL 4 => STOP);
                (IF A EQL STOPSW => STOP)
                END;    !END JSC
```

```
! FORMAT III INSTRUCTION DECODE TABLE

        FMTIII:=BEGIN

                (IF F1 EQL 0 =>
                    (DECODE F3 =>
                        JEP;                ! 50 0  JUMP ON EVEN PARITY
                        JOP;                ! 50 1  JUMP ON ODD PARITY
                        DJZ;                ! 50 2  JUMP DOUBLE PRECISION
                                            !       ZERO
                        DJNZ                ! 50 3  JUMP DOUBLE PRECISION
                                            !       NOT ZERO
                    )
                );
                (IF F1 EQL 1 =>
                    (DECODE F3 =>
                        JP;                 ! 51 0  JUMP A POSITIVE
                        JN;                 ! 51 1  JUMP A NEGATIVE
                        JZ;                 ! 51 2  JUMP A ZERO
                        JNZ                 ! 51 3  JUMP A NOT ZERO
                    )
                );
                (IF F1 EQL 2 =>
                    (DECODE F3 =>
                        LBJ;                ! 52 0  LOAD B AND JUMP
                        JBNZ;               ! 52 1  INDEX JUMP B
                        JS;                 ! 52 2  JUMP SY+B
                        JL                  ! 52 3  UNCONDITIONAL JUMP LOWER
                    )
                );
```

```
! FORMAT III INSTRUCTION DECODE TABLE (PAGE 2)

                (IF F1 EQL 3 =>
                    (DECODE F3 =>
                        BEGIN                   ! 53 0
                        (IF A EQL 0 => JNF);    !JUMP ON NO OVERFLOW
                        (IF A EQL 1 => JOF);    !JUMP ON OVERFLOW
                        (IF A GEQ 2 => OPEX)
                        END;

                        BEGIN           ! 53 1
                        DECODE A =>     !
                            JNE;        ! A=0   JUMP ON NOT EQUAL
                            JE;         ! A=1   JUMP ON EQUAL
                            JG;         ! A=2   JUMP ON GREATER THAN
                            JGE;        ! A=3   JUMP ON GREATER THAN
                                        !       OR EQUAL
                            JLT;        ! A=4   JUMP ON LESS THAN
                            JLE;        ! A=5   JUMP ON LESS THAN
                                        !       OR EQUAL
                            JNW;        ! A=6   JUMP OUTSIDE LIMITS
                            JW          ! A=7   JUMP WITHIN LIMITS
                        END;

                        BEGIN           ! 53 2
                        DECODE A =>
                            RJ;         ! A=0   RETURN JUMP
                            RJC;        ! A=1   RETURN JUMP
                            RJC;        ! A=2   RETURN JUMP
                            RJC;        ! A=3   RETURN JUMP
                            RJSC;       ! A=4   RETURN JUMP
                            RJSC;       ! A=5   RETURN JUMP
                            RJSC;       ! A=6   RETURN JUMP
                            RJSC        ! A=7   RETURN JUMP
                        END;    !END OPCODE 53 2

                        BEGIN           ! 53 3
                        DECODE A =>
                            J;          ! A=0   MANUAL JUMP
                            JC;         ! A=1   MANUAL JUMP
                            JC;         ! A=2   MANUAL JUMP
                            JC;         ! A=3   MANUAL JUMP
                            JSC;        ! A=4   MANUAL JUMP
                            JSC;        ! A=5   MANUAL JUMP
                            JSC;        ! A=6   MANUAL JUMP
                            JSC         ! A=7   MANUAL JUMP
                        END     !END OPCODE 53 3
                    )
                )
                END;    !END FORMAT III
```

```
! FORMAT IV INSTRUCTION EXECUTION

        HSCT:= ! STORE CMR IN A
                BEGIN
                (IF NOT I =>                    !HSCT - TASK STATE
        HSCT:=      BEGIN
                    A0 ← B;
                    (DECODE A =>
                    \0  TA0 ← CMR[AF4];
                    \1  TA0 ← CMR[AF4]<15:0>;
                    \2  (CKPRIV NEXT TA0 ← CMR[AF4]);
                    \3  NO.OP;
                    \4  NO.OP;
                    \5  NO.OP;
                    \6  (CKPRIV NEXT TA0 ← CMR[#60]);
                    \7  (CKPRIV NEXT TA0 ← CMR[#70])
                    ) NEXT
                    PUTA0                   ! STORE (TA0) IN ACCUMULATOR (A0)
                    END
                );
                (IF I =>                        !HSCI - INTERRUPT STATE
        HSCI:=      BEGIN
                    CKPRIV NEXT
                    A0 ← B;
                    (DECODE A =>
                    \0  TA0 ← CMR[(AF4 + #100)<6:0>];
                    \1  TA0 ← CMR[(AF4 + #100)<6:0>]<15:0>;
                    \2  TA0 ← CMR[(AF4 + #100)<6:0>];
                    \3  NO.OP;
                    \4  TA0 ← CMR[(AF4 + #100)<6:0>];
                    \5  TA0 ← CMR[(AF4 + #100)<6:0>];
                    \6  TA0 ← CMR[(AF4 + #100)<6:0>];
                    \7  TA0 ← CMR[(AF4 + #100)<6:0>]
                    ) NEXT
                    PUTA0                   ! STORE (TA0) IN ACCUMULATOR (A0)
                    END
                ) NEXT
                CC2G
                END;    !END HSCT
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 2)

        HLCTI := ! LOAD CMP FROM A
                BEGIN
                A0 ← B NEXT
                GETA0 NEXT
                (IF NOT I =>               !HLCT - TASK STATE
        HLCT :=     BEGIN
                    (DECODE A =>
                    \0  CMP[AF4] ← TA0;
                    \1  CMP[AF4] ← TA0<15:0>;
                    \2  (CKPRIV NEXT CMR[AF4] ← TA0<17:0>);
                    \3  NO.OP;
                    \4  NO.OP;
                    \5  NO.OP;
                    \6  (CKPRIV NEXT CMR[#60] ← TA0<19:0>);
                    \7  (CKPRIV NEXT CMR[#70] ← TA0<22:0>)
                    )
                    END
                );
                (IF I =>                   !HLCI - INTERRUPT STATE
        HLCI :=     BEGIN
                    CKPRIV NEXT
                    (DECODE A =>
                    \0  CMR[(AF4 + #100)<6:0>] ← TA0;
                    \1  CMR[(AF4 + #100)<6:0>] ← TA0<15:0>;
                    \2  CMR[(AF4 + #100)<6:0>] ← TA0<17:0>;
                    \3  NO.OP;
                    \4  CMR[(AF4 + #100)<6:0>] ← TA0<19:0>;
                    \5  CMR[(AF4 + #100)<6:0>] ← TA0<19:0>;
                    \6  CMR[(AF4 + #100)<6:0>] ← TA0<20:0>;
                    \7  CMR[(AF4 + #100)<6:0>] ← TA0<20:0>
                    )
                    END
                ) NEXT
                CCZC
                END;     !END HLCT
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 3)

        HLC:=   ! SHIFT LEFT CIRCULAR
                BEGIN
                SHIFTC NEXT
                GETA NEXT                 ! (AC(A)) RETURNS IN "TA0"
                TA0 ← TA0 ↑RL COUNT NEXT
                PUTA0 NEXT                ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;     !END HLC

        HDLC:=  ! SHIFT DOUBLE LEFT CIRCULAR
                BEGIN
                SHIFTC NEXT
                GETD NEXT
                TD0 ← TD0 ↑RL COUNT NEXT
                PUTD NEXT
                CC2GD
                END;     !END HDLC

        HRZ:=   ! SHIFT RIGHT FILL ZEROS
                BEGIN
                SHIFTC NEXT
                GETA NEXT                 ! (AC(A)) RETURNS IN "TA0"
                TA0 ← TA0 ↑SR0 COUNT NEXT
                PUTA0;                    ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;     !END HRZ

        HDRZ:=  ! SHIFT RIGHT DOUBLE, FILL ZEROS
                BEGIN
                SHIFTC NEXT
                GETD NEXT
                TD0 ← TD0 ↑SR0 COUNT NEXT
                PUTD;
                CC2GD
                END;     !END HDRZ

        HRS:=   ! SHIFT RIGHT SIGN FILL
                BEGIN
                SHIFTC NEXT
                GETA NEXT                 ! (AC(A)) RETURNS IN "TA0"
                (DECODE TA0<31> =>
                \0      TA0 ← TA0 ↑SR0 COUNT;
                \1      TA0 ← TA0 ↑SR1 COUNT
                ) NEXT
                PUTA0;                    ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;     !END HRS
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 4)

        HDPS:=  ! SHIFT RIGHT DOUBLE, SIGN FILL
                BEGIN
                SHIFTC NEXT
                GETD NEXT
                (DECODE TD0<63> =>
                \0      TD0 ← TD0 SR0 COUNT;
                \1      TD0 ← TD0 SR1 COUNT
                ) NEXT
                PUTD;
                CC2GD
                END;    !END HDPS

        HSF:=   ! SCALE FACTOR
                BEGIN
                GETA NEXT                  ! (AC(A)) RETURNS IN "TA0"
                TA1 ← TA0 NEXT
                (IF (TA1 EQL 0) OR (TA1 EQL ONE32) =>
                    BEGIN
                    IF A NEQ 0 =>
                        BEGIN
                        A0 ← 0 NEXT
                        TA0 ← #37 NEXT
                        PUTA0              ! STORE (TA0) IN ACCUMULATOR (A0)
                        END
                    END
                );
                (IF (TA1 NEQ 0) AND (TA1 NEQ ONE32) =>
                    BEGIN
                    COUNT ← A NEXT
                    HSF1:=BEGIN
                        IF (TA1<31> EQV TA1<30>)=>
                            (TA1 ← TA1 SL 1 NEXT
                            COUNT ← (COUNT + 1)<5:0> NEXT
                            HSF1)
                        END NEXT
                    A0 ← A NEXT
                    TA0 ← TA1 NEXT
                    PUTA0 NEXT             ! STORE (TA0) IN ACCUMULATOR (A0)
                    (IF A NEQ 0 =>
                        BEGIN
                        A0 ← 0;
                        TA0 ← COUNT NEXT
                        PUTA0              ! STORE (TA0) IN ACCUMULATOR (A0)
                        END
                    )
                    END
                ) NEXT
                TA0 ← TA1 NEXT
                CC2G
                END;    !END HSF
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 5)

        HDSF:=  ! DOUBLE SCALE FACTOR
                BEGIN
                A0 ← A NEXT
                GETD NEXT
                (IF ((TD0 EQL 0) OR (TD0 EQL ONE32@ONE32)) =>
                    BEGIN
                    IF A NEQ B =>
                        BEGIN
                        A0 ← B NEXT
                        TA0 ← #77 NEXT
                        PUTA0           ! STORE (TA0) IN ACCUMULATOR (A0)
                        END
                    END
                );
                (IF (TD0 NEQ 0) AND (TD0 NEQ ONE32@ONE32) =>
                    BEGIN
                    COUNT ← 0 NEXT
                    HDSF1:=BEGIN
                        IF (TD0<63> EQV TD0<62>)=>
                            BEGIN
                            TD0 ← TD0 SRL 1 NEXT
                            COUNT ← (COUNT + 1)<5:0> NEXT
                            HDSF1
                            END
                        END NEXT
                    A0 ← A NEXT
                    PUTD NEXT
                    (IF (A NEQ B) AND ((A + 1) NEQ B) =>
                        BEGIN
                        A0 ← B;
                        TA0 ← COUNT NEXT
                        PUTA0           ! STORE (TA0) IN ACCUMULATOR (A0)
                        END
                    )
                    END
                ) NEXT
                CC2GD
                END;    !END HDSF

        HCP:=   ! COMPLEMENT A
                BEGIN
                GETA NEXT               ! (AC(A)) RETURNS IN "TA0"
                TA0 ← NOT TA0 NEXT
                PUTA0 NEXT              ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;    !END HCP
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 6)

        HDCP:=  ! DOUBLE COMPLEMENT A
                BEGIN
                GETD NEXT
                TDD ← NOT TDD NEXT
                PUTD NEXT
                CC2G
                END;    !END HDCP


        HOR:=   ! LOGICAL SUM
                BEGIN
                A0 ← B NEXT
                GETA0 NEXT
                TA1 ← TA0 NEXT
                GETA NEXT                   ! (AC(A)) RETURNS IN "TA0"
                TA0 ← TA0 OR TA1 NEXT
                PUTA0 NEXT                  ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;    !END HOR


        HA:=    ! SUM
                BEGIN
                A0 ← B NEXT
                GETA0 NEXT
                TA1 ← TA0 NEXT
                GETA NEXT                   ! (AC(A)) RETURNS IN "TA0"
                TAC ← (TA0 + TA1) NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CC02G NEXT
                TA0 ← TAC<31:0> NEXT
                PUTA0                       ! STORE (TA0) IN ACCUMULATOR (A0)
                END;    !END HA
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 7)

        HAN:=   ! DIFFERENCE
                BEGIN
                A0 ← B NEXT
                GETA0 NEXT
                TA1 ← NOT TA0 NEXT
                GETA NEXT                 ! (AC(A)) RETURNS IN "TA0"
                TAC ← (TA0 + TA1) NEXT
                TAC ← TAC<31:0> + TAC<32> NEXT
                CC02G NEXT
                TA0 ← TAC<31:0> NEXT
                PUTA0                     ! STORE (TA0) IN ACCUMULATOR (A0)
                END;     !END HAN


        HXOR:=  ! LOGICAL DIFFERENCE
                BEGIN
                A0 ← B NEXT
                GETA0 NEXT
                TA1 ← TA0 NEXT
                GETA NEXT                 ! (AC(A)) RETURNS IN "TA0"
                TA0 ← TA0 XOR TA1 NEXT
                PUTA0 NEXT                ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;     !END HXOR


        HAND:=  ! AND
                BEGIN
                A0 ← B NEXT
                GETA0 NEXT
                TA1 ← TA0 NEXT
                GETA NEXT                 ! (AC(A)) RETURNS IN "TA0"
                TA0 ← TA0 AND TA1 NEXT
                PUTA0 NEXT                ! STORE (TA0) IN ACCUMULATOR (A0)
                CC2G
                END;     !END HAND
```

```
! FORMAT IV INSTRUCTION EXECUTION (PACE 8)

HM:=     ! MULTIPLY REGISTER
         BEGIN
         A0 ← B NEXT
         GETA0 NEXT
         TA2 ← TA0 NEXT
         GETA NEXT                    ! (AC(A)) RETURNS IN "TA0"
         SIGN ← (TA2<31> XOR TA0<31>) NEXT
         (IF TA2<31> => TA2 ← (NOT TA2));
         (IF TA0<31> => TA0 ← (NOT TA0)) NEXT
         TD0 ← TA2 * TA0 NEXT
         (IF SIGN => TD0 ← (NOT TD0)) NEXT
         TA1 ← TD0<63:32>;
         TA0 ← TD0<31:0> NEXT
         PUTA1;                       ! STORE (TA1) IN AC(A0+1)
         PUTA0 NEXT                   ! STORE (TA0) IN ACCUMULATOR (A0)
         CC2GD
         END;     !END HM


HD:=     ! DIVIDE REGISTER
         BEGIN
         A0 ← B NEXT
         GETA0 NEXT
         (IF (TA0 EQL 0) OR (TA0 EQL ONE32) =>
             CC<3> ← 1 NEXT
             BAILOUT ICYCLE
         ) NEXT
         TA2 ← TA0 NEXT
         GETD NEXT
         SIGN ← (TA1<31> XOR TA2<31>);
         SIGN1 ← (TA1<31>) NEXT
         TD0 ← (TA1@TA0) NEXT
         (IF SIGN1 => TD0 ← (NOT TD0));
         (IF TA2<31> => TA2 ← (NOT TA2)) NEXT
         TA0 ← (TD0 / TA2)<31:0> NEXT
         TAC ← (TD0 MINUS (TA0 * TA2))<31:0> NEXT
         TA1 ← (TAC<31:0> + TAC<32>)<31:0> NEXT
         (IF SIGN => TA0 ← (NOT TA0));
         (IF SIGN1 => TA1 ← (NOT TA1)) NEXT
         PUTA1;                       ! STORE (TA1) IN AC(A0+1)
         PUTA0;                       ! STORE (TA0) IN ACCUMULATOR (A0)
         CC2G;
         CC<3> ← (TD0<63:31> NEQ 0)
         END;     !END HD
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 9)

        HRT:=   ! SQUART ROOT
                BEGIN
                CC<3:1> ← 0;
                GETD NEXT
                (IF TD0<63> => CC<3> ← 1 NEXT BAILOUT ICYCLE) NEXT
                TDAC ← TD0 NEXT
                TD0 ← 0;
                TD1 ← 0;
                TD2 ← 0 NEXT
                TD0<62> ← 1;
                TD2<63> ← 1;
                COUNT ← 32 NEXT
                HRTLP:= BEGIN
                        IF COUNT =>
                            BEGIN
                            (IF TDAC GEQ (TD0 + TD1)<63:0> =>
                                TDAC ← (TDAC MINUS (TD0 + TD1))<63:0> NEXT
                                TD1 ← (TD1 + TD2)<63:0>
                            ) NEXT
                            TD0 ← TD0 !SR0 2;
                            TD1 ← TD1 !SR0 1;
                            TD2 ← TD2 !SR0 2;
                            COUNT ← (COUNT MINUS 1)<5:0> NEXT
                            HRTLP
                            END
                        END NEXT
                        A0 ← 0;
                        TD0<31:0> ← TD1<31:0>;
                        TD0<63:32> ← TDAC<31:0> NEXT
                        PUTD NEXT
                        (IF TDAC<63:32> => CC<3> ← 1);
                        (IF (TD0 EQL 0) OR (TD0 EQL ONE32@ONE32) =>
                                RCC<2:1> ← 3)
                END;    !END HRT


        HLB:=   ! LOAD BA WITH B0
                BEGIN
                IF A NEQ 0 =>
                    B0 ← B NEXT
                    GETB NEXT           ! INDEX REGISTER RETURNS IN "TB"
                    TA0 ← TB NEXT
                    B0 ← A NEXT
                    GETB NEXT
                    TBD ← TA0<15:0> NEXT
                    PUTB                ! STORE (TB) IN INDEX REGISTER (B0)
                END;    !END HLB
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 10)

        HC:=    ! COMPARE, REGISTER
                BEGIN
                COMPAR ← 1;
                A0 ← B NEXT
                GETA0 NEXT
                TA1 ← TA0 NEXT
                GETA NEXT                       ! (AC(A)) RETURNS IN "TA0"
                (IF TA0 EQL ONE32 => TA0 ← 0);
                (IF TA1 EQL ONE32 => TA1 ← 0);
                CC<2:1> ← 0 NEXT
                TSTR ← (TA0 TST TA1) NEXT
                (IF (TA0<31> XOR TA1<31>) => TSTR ← (2 - TSTR)<1:0>) NEXT
                (DECODE TSTR =>
                \0      NO.OP;
                \1      CC<2:1> ← 3;
                \2      CC<1> ← 1;
                \3      NO.OP)
                END;    !END HC

        HCL:=   ! COMPARE LIMITS, REGISTER
                BEGIN
                COMPAR ← 1;
                A0 ← B NEXT GETA0 NEXT
                TA2 ← TA0 NEXT
                GETD NEXT
                (IF TA0 EQL ONE32 => TA0 ← 0);
                (IF TA1 EQL ONE32 => TA1 ← 0);
                (IF TA2 EQL ONE32 => TA2 ← 0);
                CC<0> ← 0 NEXT
                (IF (TA2 GEQ TA1) OR (TA0 GTR TA2) => CC<0> ← 1)
                END;    !END HCL

        HCM:=   ! COMPARE MASKED, REGISTER
                BEGIN
                COMPAR ← 1;
                A0 ← B NEXT
                GETA0 NEXT
                TA2 ← TA0 NEXT
                GETD NEXT
                TA1 ← TA1 AND TA0 NEXT
                (IF TA2 EQL ONE32 => TA2 ← 0);
                (IF TA1 EQL ONE32 => TA1 ← 0);
                CC<2:1> ← 0 NEXT
                TSTR ← (TA1 TST TA2) NEXT
                (IF (TA1<31> XOR TA2<31>) => TSTR ← (2 - TSTR)<1:0>) NEXT
                (DECODE TSTR =>
                \0      NO.OP;
                \1      CC<2:1> ← 3;
                \2      CC<1> ← 1;
                \3      NO.OP)
                END;    !END HCM
```

```
! FORMAT IV INSTRUCTION EXECUTION (PAGE 11)

        HCB:=   ! COMPARE BB WITH BA
                BEGIN
                COMPAR ← 1;
                BO ← B NEXT
                GETB NEXT               ! INDEX REGISTER RETURNS IN "TB"
                TAO ← TB NEXT
                BO ← A NEXT
                GETB NEXT               ! INDEX REGISTER RETURNS IN "TB"
                (IF TAO<15:0> EQL "FFFF => TAO ← 0);
                (IF TBD<15:0> EQL "FFFF => TBD ← 0) NEXT
                TSTR ← (TAO<15:0> TST TBD<15:0>) NEXT
                (IF TAO<15> XOR TBD<15> => TSTR ← (2 - TSTR)<1:0>) NEXT
                (DECODE TSTR =>
                \0      CC<2:1> ← 0 ;
                \1      CC<2:1> ← 3 ;
                \2      CC<2:1> ← 1 ;
                \3      CC<2:1> ← 0 )
                END;    !END HCB

        HSIM:=  ! STORE IOC MONITOR CLOCK IN A
                BEGIN                   !**NOT IMPLEMENTED**
                NO.OP
                END;    !END HSIM

        HSTC:=  ! STORE REAL-TIME CLOCK IN A
                BEGIN                   !**NOT IMPLEMENTED**
                NO.OP
                END;    !END HSTC

        HPI:=   ! PREVENT CLASS III INTERRUPTS
                BEGIN
                CKPRIV NEXT
                CLASLO<3> ← 1
                END;    !END HPI

        HAI:=   ! ALLOW CLASS III INTERRUPTS
                BEGIN
                CKPRIV NEXT
                CLASLO<3> ← 0
                END;    !END HAI

        HALT:=  ! STOP PROCESSOR
                BEGIN
                STOP
                END;    !END HALT

        HWFI:=  ! WAIT FOR INTERRUPT
                BEGIN
                HWFLAG ← 1 NEXT
                HWFLP:= BEGIN
                        INT NEXT
                        (IF HWFLAG => HWFLP)
                        END
                END;    !END HWFI
```

```
! FORMAT IV INSTRUCTION DECODE TABLE

        FMTIV:= BEGIN
                UPLOW = NOT UPLOW;
                (IF F0 EQL 6 =>
                    BEGIN
                    DECODE F1 =>
                        HSCTI;              ! 60   HSCT IF I=0,
                                            !      HSCI IF I=1
                        HLCTI;              ! 61   HLCT IF I=0,
                                            !      HLCI IF I=1
                        HLC;                ! 62   SHIFT LEFT CIRCULARLY
                        HDLC;               ! 63   SHIFT LEFT CIRCULARLY
                                            !        DOUBLE
                        HRZ;                ! 64   SHIFT RIGHT FILL ZEROS
                        HDRZ;               ! 65   SHIFT RIGHT DOUBLE,
                                            !        FILL ZEROS
                        HRS;                ! 66   SHIFT RIGHT FILL SIGN
                        HDRS                ! 67   SHIFT RIGHT DOUBLE,
                                            !        FILL SIGN
                    END          !END OPCODE 6X
                );

                (IF F0 EQL 7 =>
                    (DECODE F1@F2 =>
                        HSF;                ! 70 0  SCALE FACTOR
                        HDSF;               ! 70 1  DOUBLE SCALE FACTOR
                        HCP;                ! 70 2  COMPLEMENT A
                        HDCP;               ! 70 3  DOUBLE COMPLEMENT A
                        OPEX;               ! 70 4
                        OPEX;               ! 70 5
                        OPEX;               ! 70 6
                        OPEX;               ! 70 7
                        HOR;                ! 71 0  LOGICAL SUM
                        HA;                 ! 71 1  SUM
                        HAN;                ! 71 2  DIFFERENCE
                        HXOR;               ! 71 3  LOGICAL DIFFERENCE
                        OPEX;               ! 71 4
                        HAND;               ! 71 5  AND
                        OPEX;               ! 71 6
                        OPEX;               ! 71 7
                        OPEX;               ! 72 0
                        OPEX;               ! 72 1
                        OPEX;               ! 72 2
                        OPEX;               ! 72 3
                        OPEX;               ! 72 4
                        OPEX;               ! 72 5
                        OPEX;               ! 72 6
                        OPEX;               ! 72 7
                        OPEX;               ! 73 0
                        OPEX;               ! 73 1
                        OPEX;               ! 73 2
! FORMAT IV INSTRUCTION DECODE TABLE (PAGE 2)
                        OPEX;           ! 73 3
                        OPEX;           ! 73 4
                        OPEX;           ! 73 5
                        OPEX;           ! 73 6
                        OPEX;           ! 73 7
                        HM;             ! 74 0  MULTIPLY REGISTER
                        HD;             ! 74 1  DIVIDE REGISTER
                        HRT;            ! 74 2  SQUARE ROOT
                        HLB;            ! 74 3  LOAD B(A) WITH B(B)
                        HC;             ! 74 4  COMPARE REGISTER
                        HCL;            ! 74 5  COMPARE LIMITS, REGISTER
                        HCM;            ! 74 6  COMPARE MASKED, REGISTER
                        HCB;            ! 74 7  COMPARE B(B) WITH B(A)
                        OPEX;           ! 75 0
                        OPEX;           ! 75 1
                        OPEX;           ! 75 2
                        OPEX;           ! 75 3
                        OPEX;           ! 75 4
                        OPEX;           ! 75 5
                        OPEX;           ! 75 6
                        OPEX;           ! 75 7
                        OPEX;           ! 76 0
                        OPEX;           ! 76 1
                        OPEX;           ! 76 2
                        OPEX;           ! 76 3
                        OPEX;           ! 76 4
                        OPEX;           ! 76 5
                        OPEX;           ! 76 6
                        OPEX;           ! 76 7
                        HSIM;           ! 77 0  STORE IOC MONITOR
                                        !         CLOCK IN A
                        HSTC;           ! 77 1  STORE REAL-TIME CLOCK
                                        !         IN A
                        OPEX;           ! 77 2
                        OPEX;           ! 77 3
                        HPI;            ! 77 4
                        HAI;            ! 77 5  ALLOW CLASS III INTERRUPT
                        (DECODE I =>
                            HALT;       ! I=0   STOP PROCESSOR
                            HWFI        ! I=1   WAIT FOR INTERRUPT
                        );
                        OPEX            ! 77 7
                    )
                )
                END;    !END FORMAT IV
```

```
! INSTRUCTION EXECUTION

        IEXEC:= BEGIN
                (IF UFLOW => (FMTIV NEXT BAILOUT ICYCLE)) NEXT
                (DECODE F0 =>
                \0  FMTIII;
                \1  FMTI;
                \2  FMTI;
                \3  FMTI;
                \4  FMTI;
                \5  BEGIN
                    (IF F1 LEQ #3 => FMTIII);
                    (IF F1 GEQ #4 => FMTI)
                    END;
                \6  FMTIV;
                \7  FMTIV
                )
                END;     !END IEXEC

! INSTRUCTION CYCLE

        ICYCLE:=BEGIN
                (IF NOT RPFLAG => READIN) NEXT
                IEXEC NEXT
                (IF RPFLAG => CKRPT) NEXT
                (IF EXRF => (IEXEC NEXT EXRF ← 0))
                END

        ERALCED !END OF DECLARATIONS

! MAIN EXECUTABLE PROGRAM

        RUN:=   BEGIN
                IF NOT STOPBIT =>
                    ICYCLE NEXT
                    (IF NOT ILOCK => INT) NEXT
                    RUN
                END     !END OF RUN LOOP

! END OF AN/UYK-7
)
```

3. AN/GYK-12 ISPL Description

```
GYK12.ISP[K710CA001]@CMU-10D      23-Feb-78 11:30      WD15@CMU-10D      Page 1-1

GYK12:=
        (DECLARE

! This ISPL description based on the following publication:
!       AN/GYK-12 Computer Principles of Operation Manual
!       Programming Support System (Litton Data Systems)
!       29 August 1977
!       USACSCS-TF-4-3
            ;
        MACRO BEGIN:= ( $
        MACRO END  := ) $
        MACRO PLUS := MINUS (MINUS $     !This is 2's comp. addition
        MACRO PLEN := ) $                !this should consider as blank

        MACRO Maxw := 32767 $   !The architecture supports 33,554,432 words,
        MACRO Maxh := 65535 $   !or 67,108,864 halfwords,
        MACRO Maxb :=131071 $   !or 1,073,741,824 bytes.


!Note the following:
!       Architecture related variables appeared in upper case
!         throughout the ISPL description, as compare with
!         implementation related variables are in lower case.


!Primary memory (organized in 16K pages of 2K of 32 bit words)
!--------------

        MEMW[0:maxw]<0:31>;                              !Word memory
        MEMH[0:maxh]<0:15>      := MEMW[0:maxw]<0:31>;  !Halfword memory
        MEMB[0:maxb]<0:7>       := MEMW[0:maxw]<0:31>;  !Byte Memory


        !This is the low end of the primary memory.  It is used to
        !  store the state of idle program levels.
        BASEPAGE[0:2047]<0:31>  := MEMW[0:2047]<0:31>;
        HBASEPAGE[0:4095]<0:15> := MEMH[0:4095]<0:15>;


!Processor state ---  The GYK-12 hardware manages many multiprogramming
!       tasks left to software in other systems.  There are 64 separate
!       hardware "program  levels" ; level 63  has the lowest priority,
!       and level  0  the  highest.  Each  level  has its  own  storage
!       locations in the base page  of primary memory to keep copies of
!       most of the following registers.
!
!
!The  following  registers  fall  into  the  GYK12  "special  address"
!category.  They  are accessed by halfword operand  addresses "00 - "3F
!in the order given below.   Note that they are not mapped into primary
!memory.  Copies stored in the BASEPAGE of memory are inactive.


!Special Address
!---------------

!\p - Locations that can only be accessed by a program level with "privelege"
!\s - Indicates that only a single copy of this register exists.  All others
!     have one copy per program level stored in the base page of memory.

!(1)
\0.1F    PREGW[0:15]<0:31>;                         !Fast general process registers
         PREGH[0:31]<0:15> := PREGW[0:15]<0:31>;   !the same
\0           INDREG<0:15>:=PREGW[0]<0:15>;          !Indicator register - flags, etc.
                 OT<> := INDREG<0>;                 !Overflow trap override
                 LC<> := INDREG<1>;                 !Level change
!                     := INDREG<2:5>                !Spare bits
                 MT<> := INDREG<6>;                 !Memory test
                 PV<> := INDREG<7>;                 !Privelege violation
                 IE<> := INDREG<8>;                 !Input parity error
                 ME<> := INDREG<9>;                 !Memory parity eror
                 DT<> := INDREG<10>;                !Device timeout
                 NF<> := INDREG<11>;                !Non-implemented instruction flag
                 CF<> := INDREG<12>;                !Carry flag
                 OF<> := INDREG<13>;                !Overflow greater flag
                 EF<> := INDREG<14>;                !Equal/excess flag
                 LF<> := INDREG<15>;                !Less flag
                 FLAGS<0:3> := INDREG<12:15>;       !Condition code flags
\1           PC<0:15> := PREGW[0]<16:31>;           !Program counter (active)
                                                    !PC<15> always zero
\1C.1D       MASKREG<0:31> := PREGW[14]<0:31>;      !Mask register
\1E.1F       TRAPREG<0:31> := PREGW[15]<0:31>;      !Instr to be exec during an trap
\20.2F
\p       PCAR[0:15]<0:15> ;                         !Page control & Address Registers
\30.31
\p       PLLREG<0:31> ;                             !Privelege and level link register
```

```
            PP<0:1>      := PLLREG<0:1>;             !priveleqe of active process
            LPL<0:5>     := PLLREG<2:7>;             !Link program level
            C<0:1>       := PLLREG<8:9>;             !Level control (control loading &
                                                     !storing of registers in response
                                                     !to interrupts.)
            CPL<0:5>     := PLLREG<10:15>;           !call program level (To)
            LA<0:15>     := PLLREG<16:31>;           !link argument (transmit info to
                                                     !called p.l.)
\32.33  QREG<0:31>;                                  !Queue register
                                                     !exists in main memory
\34.35
\p \m   QUERYREG<0:31>;                              !query register
            IPE<>        := QUERYREG<0>;             !Instruction parity error
            LPE<>        := QUERYREG<1>;             !Level change parity error
            IV<>         := QUERYREG<2>;             !instruction violation
            MV<>         := QUERYREG<3>;             !Memory access violation
            MTO<>        := QUERYREG<4>;             !Memory time out
            SV<>         := QUERYREG<5>;             !Specification violation
            QD<0:3>      := QUERYREG<6:9>;           !Page designator
            PPL<0:5>     := QUERYREG<10:15>;         !prior program level
!                        := QUERYREG<16:31>;         !These bit are read only
            LLI<>        := QUERYREG<16>;            !level lock indicator
            EE<>         := QUERYREG<17>;            !error exit
            TPL<0:5>     := QUERYREG<18:23>;         !tentative program level
!                        := QUERYREG<24:25>          !Spare bits
            APL<0:5>     := QUERYREG<26:31>;         !active program level
\36
\p \m   ELR<0:15>;                                   !Executive link register
            XPL<0:5>:= ELR<10:15>;                   !Executive program level called
\37
\p      Notused<0:15>;                               !Unused
\38.3f
\p \m   PAR[0:7]<0:15>;                              !Program activity registers -
                                                     !These keep a record of active
                                                     !and suspended program levels
                PS1[0:15]<>:= PAR[0]<0:15>;          !make the above bit addressable
                PS2[0:15]<>:= PAR[2]<0:15>;
                PS3[0:15]<>:= PAR[4]<0:15>;
                PS4[0:15]<>:= PAR[6]<0:15>;
                PE1[0:15]<>:= PAR[1]<0:15>;
                PE2[0:15]<>:= PAR[3]<0:15>;
                PE3[0:15]<>:= PAR[5]<0:15>;
                PE4[0:15]<>:= PAR[7]<0:15>;


!Architecture related register visible to programmer indirectly.

        PLLFF<>;          !program level lock same as LLI except in
                          !program level two.
        LOCKPCAR<>;       !restrict access to memory through PCAR4-15
                          !result from short program level change
        CTS<0:3>;         !Computer Test Set(Conditional transfer switches)
```

```
!I/O RELATED REGISTERS
!The I/O  device command  and status  words are  also stored  in  fixed
!locations  in the BASEPAGE of  memory.  The mapping will  not be given
!here,  however,  since   these  words  are  scattered  throughout  the
!BASEPAGE.    Suffice it  to say  that each device  has a  keyword and a
!termination  word, the  bit  assignments  of which  are  given  below.
!Device  "01 is reserved  for a  monitor register whose  function is to
!contain pertinent information during an I/O error interrupt.

        monitor<0:31>:              !read through device 01,
                                    !set through device address 01,
                                    !reset through device address 02.
        maintreg<0:31>:             !read through device 02

!Bit layout of monitor register:

MACRO  sysfault:= 1 $               !System fault
MACRO  comfault:= 2 $               !Computer fault
MACRO  io.error:= 7 $               !Input or output error
MACRO  diperror:= 8 $               !Device input parity error
MACRO  dtoerror:= 9 $               !Device timeout error
MACRO  memerror:= 10 $              !memory access violation during I/O
MACRO  cwperror:= 11 $              !control word parity error
MACRO  dwperror:= 12 $              !data word parity error
MACRO  mtoerror:= 13 $              !Memory timeout error
MACRO  iocerror:= 14 $              !I/O Controller time out error
MACRO  ptoerror:= 15 $              !Program time out error
MACRO  av.error:= 16 $              !I/O avalanche error (multiple I/O errors)
MACRO  memadd  := 21:24 $                  !Most significant bits of memory address
                                    !during I/O error.
MACRO  devadd  := 25:31 $                  !Device address during I/O error.


!Architectural features of I/O not supported here:
!
!-I/O controller
!-I/O memory access control - associated with device address "00
!-Maintenance panel controls - associated with device address "02
 -Real time clocks - associated with device addresses "03-"05


!Bit assignments in termination word:

MACRO blockcomplete:= 0 $
MACRO interupt:= 1 $
MACRO npl:= 2:7 $                   !normal return program level
MACRO transerror:= 8 $
MACRO operror := 9 $
MACRO epl:= 10:15 $                 !error return program level
MACRO chnend:= 16 $
MACRO parterm:= 17 $                !parity termination
MACRO qtc:= 18:23 $                 !queue table control - controls entry in
                                    !queue regusister upon completion of task
MACRO devsta := 24:31 $             !device status


!Bit assignments in keyword:

MACRO blocklength := 0:10 $
MACRO iomode := 11:13 $             !I/O modes:      0-inactive
                                    !                1-output, full word by bytes
                                    !                2-alarm (clock)
                                    !                3-input, full word by bytes
                                    !                4-output, upper byte in halfword
                                    !                5-output,lower byte in halfword
                                    !                6-input, upper byte in halfword
                                    !                7-input, lower byte in halfword
MACRO curadd := 14:31 $             !current address


        IOdevice[0:127]<0:7>:               !An array which pretends to be afull
                                            !complement of I/O devices.  It accepts
                                            !device commands.
        IOports[0:127]<0:8>:                !This array pretends to be the I/O
                                            !data ports associated with the
                                            !above devices
        IDEVNM<0:7>:                        !number of dev requesting int

! Maintenance codes and status
        dincode<17:0>:
        cpuiou<1:0>:
```

```
[illegible]
!          These variables will appear in lower case throughout the ISP.


        ir<0:31>,                     !Instruction Register
        irshd<0:31>:=ir<0:31>,        !"shadow register" to be counted
                                       for architecture measures
            opsize<>:=ir<0>,          !Operand size - 0=>fullword, 1=>halfword
            opcode<0:6>:=ir<0:6>,
            admod<0:1>:=ir<7:8>,              !addressing mode
            accum<0:3>:=ir<9:12>,             !accumulator selection
            index<0:2>:=ir<13:15>,            !index register selection
            immed<0:7>:=ir<8:15>,             !immediate operand field
            opadd<0:15>:=ir<16:31>,           !operand/address
            d<0:3>:=opadd<0:3>,               !page designator
            e<0:10>:=opadd<4:14>,             !word address offset
            w<>:=opadd<15>,                   !halfword address bit
        effadd<0:15>,                 !Effective address
        byteselect<>,                 !0=upper byte; 1=lower byte
        mar<0:25>,                    !Memory Address Register
        mbr<0:31>,                    !Memory data register
        srctmp<0:32>,                 !Temporary source register
           bitsrc[0:15]<>:=srctmp<17:32>,
        dsttmp<0:32>,                 !Temporary destination register
           bitdst[0:15]<>:=dsttmp<17:32>,
        pregtmp<0:32>,                !Temporary process register
        pregd<0:63>,                  !Temp double register
        pronly<>,                     !special addresses limited to process
                                      !registers
        exmode<>,                     !Addressing mode 0 excluded
        noextend<>,                   !flag to inhibit sign extension during
                                      !operand fetch
        notrap<>,                     !flag set to allow overflow
                                      !without trap in mode 0 with indexing
        trapflag<>,                   !set when trap condition exists
        noflags<>,                    !flag to inhibit condition code setting
                                      !during mode 0 indexing
        cfonly<>,                     !set only CF during mode 0 indexing
        execute<>,                    !flag set during execute instruction
                                      !to repeat iexec routine
        tmpctr<0:7>,                  !temporary counter
        newpl<0:5>,                   !new program level
        partmp[0:3]<0:15>,            !temporary program activity register
                                      ! - used to select new level
        auctmp[0:63]<> := partmp[0:3]<0:15>;    !makes above bit addressable

        intflag<>,                    !set by the I/O controller when it
                                      !wants normal service
        interr<>,                     !set by the I/O controller to indicate
                                      !an error when no error level was
                                      !specified by software
        noopreq<>,
        MACRO noop:=noopreq = 0 &,
        stopbit<>,                    !a flag to stop the simulation
        tmp66<65:0>,                  !66 bit temporary
        tmpflag<>,                    !one bit temporary
        t1<>,                         !ditto
        t1a<>,                        !ditto
        t4<0:3>,                      !four bit temporary
        t64<63:0>,                    !64 bit temporary
        tmp33<32:0>,                  !33 bit temporary
        alu33<0:32>,                  !33 bit from adder
        t32<0:31>,                    !32 bit temporary
        bitt32[0:31]<> := t32<0:31>;        !makes above bit addressable
        brkpnt<[illegible]> !breakpoint address
                   ! remove this from isp after debug


!************************************
! io controller
IOC :=    BEGIN    !I/O Controller - this is the place to insert a
                   !declaration of a asynchronous process which
                   !automatically interrupts the cpu at desired intervals
!         [illegible] monitor
          [illegible] next
          [illegible] next
          [illegible] and [illegible] interrupt[illegible]
                   [illegible] and [illegible] next
                   [illegible] next
          [illegible]
                   [illegible]
                   [illegible]
                   [illegible]
                   [illegible]
          )
          END
```

```
!UTILITY ROUTINES


!Hardware utilities for "program level" changes!



!Save all process registers

savregs:= BEGIN
          tmpctr ← 0 NEXT
     sloop:= BASEPAGE[(APL * "20) + tmpctr)<10:0>] ← PREGH[tmpctr<4:7>] NEXT
               tmpctr ← (tmpctr + 1)<7:0> NEXT
               (IF tmpctr LEQ 15 => sloop)
          END;


!Save process register 0 only

savr0:= BEGIN
          BASEPAGE[(APL * "20)<10:0>] ← PREGH[0]
          END;


!Load all process registers

loadregs:=BEGIN
          tmpctr ← 0 NEXT
     lloop:= PREGH[tmpctr<4:7>] ← BASEPAGE[((APL * "20) + tmpctr)<10:0>] NEXT
               tmpctr ← (tmpctr + 1)<7:0> NEXT
               (IF tmpctr LEQ 15 => lloop) NEXT
          LC ← 1
          END;


!Load register 0 only

loadr0:= BEGIN
          PREGH[0] ← BASEPAGE[(APL * "20)<10:0>] NEXT
          LC ← 1
          END;



!Load page control access registers

loadpcar:=BEGIN
          tmpctr ← 0 NEXT
     ldloop:=PCAR[tmpctr<4:7>] ← MEMH[(((APL * "40) + "20) + tmpctr)<11:0>] NEXT
               tmpctr ← (tmpctr + 1)<7:0> NEXT
               (IF tmpctr LEQ 15 => ldloop)
          END;


!Load first 4 PCAP only

load4pcar:=BEGIN
          tmpctr ← 0 NEXT
     ld4loop:=PCAP[tmpctr<4:7>] ← MEMH[(((APL * "40) + "20) + tmpctr)<11:0>] NEXT
               tmpctr ← (tmpctr + 1)<7:0> NEXT
               (IF tmpctr LEQ 3 => ld4loop)
          END;


!select highest priority program level (result in newpc)

auction:= BEGIN
          auctmp[0] ← PAP[0] AND PAP[1];
          auctmp[1] ← PAP[2] AND PAP[3];
          auctmp[2] ← PAP[4] AND PAP[5];
          auctmp[3] ← PAP[6] AND PAP[7] NEXT
          newpl ← 0 NEXT
          aucloop:=(IF auctmp[newpl] => RAIL(OU) auction) NEXT
                    newpl ← (newpl + 1)<5:0> NEXT
                    (IF newpl LSS 63 => aucloop)
          END;


!Swap to program level two (in response to error conditions)

swaptwo:=BEGIN
          (IF [illegible] => RAIL(OU) cycle) NEXT
          savr0 NEXT
          PPN ← APN NEXT
          [illegible] ← BASEPAGE["58] NEXT
          APN ← 2 NEXT
```

```
        (DECODE C =>
                (loadregs; loadpcar ; LOCKPCAR ← 0);
                (loadregs; load4pcar; LOCKPCAR ← 1);
                (loadr0; loadpcar ; LOCKPCAR ← 0);
                (loadr0; load4pcar; LOCKPCAR ← 1)
        );
        PAP[0]<2> ← 1;
        PLLFF ← 1 NEXT          !note LLT unchanged
        BAILOUT icycle          !*Don't look at interrupts - go to level 2
        END;


!swap to new program level in response to interrupt or level change instruction

swap:=  BEGIN
        auction NEXT            !select new program level
        (DECODE LC =>           !software initiated p.l. change
                savregs;
                savr0
        ) NEXT
        PPL ← APL NEXT
        APL ← newpl NEXT
        PLLPEG ← BASEPAGE[((newpl * '20) + '10)<10:0>] NEXT
        (DECODE C=>
                (loadregs; loadpcar ; LOCKPCAR ← 0);
                (loadregs; load4pcar; LOCKPCAR ← 1);
                (loadr0; loadpcar ; LOCKPCAR ← 0);
                (loadr0; load4pcar; LOCKPCAR ← 1)
        )
        END;
```

```
!Privelege and access checking utilities

!Operation unspecified by the architecture
!        -- no warning is given to programmer
!        -- behavior like no-operation, but register might be changed

unspec:=BEGIN
        nop
        END;


!Privelege violation action

pviolation:= BEGIN
        PV ← 1 NEXT
        wuwntwo
        END;


!NON-implemented instruction (as defined by architecture)

noli=   BEGIN              !unused instruction code
        NF ← 1 NEXT
        trapflag ← 1
        END;


!Check memory access violation

setmv:= BEGIN
        (DECODE APL EQL 2 =>
                MV ← 1;
                EE ← 1) NEXT
        pviolation
        END;


!Check for "read data" access to page

ckrda:= BEGIN
        IF (PCAR[effadd<0:3>]<0:1> EQL 3) OR
           (LOCKPCAR AND (effadd<0:3> GTR 3)) =>
                setmv
        END;


!check for "read instruction" access to page

ckria:= BEGIN
        IF (PCAR[effadd<0:3>]<0:1> GEQ 1) OR
           (LOCKPCAR AND (effadd<0:3> GTR 3)) =>
                setmv
        END;


!check for write access to page

ckwa:=  BEGIN
        IF (PCAR[effadd<0:3>]<0:1> NEQ 0) OR
           (LOCKPCAR AND (effadd<0:3> GTR 3)) =>
                setmv
        END;


!Check instruction violation

setiv:= BEGIN
        (DECODE APL EQL 2 =>
                IV ← 1;
                EE ← 1) NEXT
        pviolation
        END;


!Check for privelege status of current active program level

prvchk:=(IF PR NEQ '10 => setiv);


!Check for semi-priveleged status

sprivchk:=(IF (PP EQL 0) OR (PP EQL 3) => setiv);


!check read special address access privelege
```

```
ckrspa:=BEGIN
        (IF effadd<10> =>        !EXF opcode = "2E
            (IF pronly OR ((opcode EQL "2E) AND (PR NEQ '10)) =>
                setiv
            )
        ) ;
        END;

!check write special address access privlege

ckwspa:=BEGIN
        (IF effadd<10> =>        !addr of queue reg = 25
            (IF pronly OR ((effadd<1:14> NEQ 25) AND (PR NEQ '10)) => setiv)
        )
        END;
```

```
!Operand read and write utilities


!Translate virtual to real address

virt.real:=BEGIN
        mar ← PCAP[effadd<0:3>]<2:15>@effadd<4:15>
        END;

!Read from special address (halfword)

rsph:=  BEGIN
        ckrspa NEXT
        (DECODE effadd<10:11> =>
        \0      srctmp ← PREGH[effadd<11:15>] ;
        \1      srctmp ← PREGH[effadd<11:15>] ;
        \2      (srctmp ← PCAP[effadd<12:15>] NEXT noextend ← 1);
        \3      (DECODE effadd<12:15> =>
                \0      srctmp ← HBASEPAGE[(APL*"40+"30)<11:0>];
                \1      srctmp ← HBASEPAGE[(APL*"40+"31)<11:0>];
                \2      srctmp ← HBASEPAGE[(APL*"40+"32)<11:0>];
                \3      srctmp ← HBASEPAGE[(APL*"40+"33)<11:0>];
                \4      srctmp ← QUERYREG<0:15>;
                \5      srctmp ← QUERYREG<16:31>;
                \6      srctmp ← ELP;
                \7      srctmp ← notused;
                \8      srctmp ← PAR[0];
                \9      srctmp ← PAR[1];
                \A      srctmp ← PAR[2];
                \B      srctmp ← PAR[3];
                \C      srctmp ← PAR[4];
                \D      srctmp ← PAR[5];
                \E      srctmp ← PAR[6];
                \F      srctmp ← PAR[7]
                )
        )
        END;  !of rsph


!Read special address (full word)

rspw:=  BEGIN
        ckrspa NEXT
        (DECODE effadd<10:11> =>
        \0      srctmp ← PREGW[effadd<1?:14>] ;
        \1      srctmp ← PREGW[effadd<11:14>] ;
        \2      srctmp ← PCAP[effadd<12:15>]  ; !note halfword result
        \3      (DECODE effadd<12:14> =>
                \0      srctmp ← BASEPAGE[(APL*"20+"18)<10:0>];
                \1      srctmp ← BASEPAGE[(APL*"20+"19)<10:0>];
                \2      srctmp ← QUERYREG;
                \3      srctmp ← ELP @ notused;
                \4      srctmp ← PAR[0] @ PAR[1];
                \5      srctmp ← PAR[2] @ PAR[3];
                \6      srctmp ← PAR[4] @ PAR[5];
                \7      srctmp ← PAR[6] @ PAR[7]
                )
        )
        END;    !of rspw


!Write to a special address (halfword)

wsph:=  BEGIN
        ckwspa NEXT
        (DECODE effadd<10:11> =>
        \0      PREGH[effadd<11:15>] ← dsttmp<17:32> ;
        \1      PREGH[effadd<11:15>] ← dsttmp<17:32> ;
        \2      (PCAP[effadd<12:15>] ← dsttmp<17:32> ;
                 HBASEPAGE[(APL*"40+"20+effadd<12:15>)<11:0>] ← dsttmp<17:32>
                );
        \3      (DECODE effadd<12:15> =>
                \0      (HBASEPAGE[(APL*"40+"30)<11:0>] ← dsttmp<17:32>;
                         PLLREG<0:15> ← dsttmp<17:32>);
                \1      (HBASEPAGE[(APL*"40+"31)<11:0>] ← dsttmp<17:32>;
                         PLLREG<16:31> ← dsttmp<17:32>);
                \2      HBASEPAGE[(APL*"40+"32)<11:0>] ← dsttmp<17:32>;
                \3      HBASEPAGE[(APL*"40+"33)<11:0>] ← dsttmp<17:32>;
                \4      QUERYREG<0:15> ← dsttmp<17:32>;
                !The least significant half of the QUERYREG can never
                !be modified by software, but if a M2H instruction
                !addresses this least significant half, it clears the
                !most significant half. (Cf. sec. 6-3S)
                \5      (If opcode EQL 67 => QUERYREG<0:15> ← 0);
                \6      ELP ← dsttmp<17:32>;
                \7      notused ← dsttmp<17:32>;
                \8      PAR[0] ← dsttmp<17:32>;
                \9      PAR[1] ← dsttmp<17:32>;
                \A      PAR[2] ← dsttmp<17:32>;
                \B      PAR[3] ← dsttmp<17:32>;
                \C      PAR[4] ← dsttmp<17:32>;
                \D      PAR[5] ← dsttmp<17:32>;
                \E      PAR[6] ← dsttmp<17:32>;
                \F      PAR[7] ← dsttmp<17:32>
                )
        )
        END;    !of wsph


!Write to a special address(full word)

wspwi:= BEGIN
        ckwspa NEXT
        (DECODE effadd<10:11> =>
        \0      PREGW[effadd<11:14>] ← dsttmp<1:32> ;
        \1      PREGW[effadd<11:14>] ← dsttmp<1:32> ;
        \2      (PCAP[effadd<12:15>] ← dsttmp<17:32> ;
                 HBASEPAGE[(APL+"40+"20+effadd<12:15>)<11:0>] ← dsttmp<17:32>
                );
        \3      (DECODE effadd<12:14> =>
                \0      (BASEPAGE[(APL+"20+"18)<10:0>] ← dsttmp<1:32>;
                         PLLPEG ← dsttmp<1:32>);
                \1      BASEPAGE[(APL+"20+"19)<10:0>] ← dsttmp<1:32>;
                !see comment on wsph. (CF. sec. 6-39)
                \2      QUERYPEG ← dsttmp<1:16> @ QUERYPEG<16:31>;
                \3      (ELP ← dsttmp<1:16>; notused ← dsttmp<17:32>);
                \4      (PAR[0] ← dsttmp<1:16>; PAR[1] ← dsttmp<17:32>);
                \5      (PAR[2] ← dsttmp<1:16>; PAR[3] ← dsttmp<17:32>);
                \6      (PAR[4] ← dsttmp<1:16>; PAR[5] ← dsttmp<17:32>);
                \7      (PAR[6] ← dsttmp<1:16>; PAR[7] ← dsttmp<17:32>)
                )
        )
        END;
```

```
!Read data from a halfword

rdhw:=  BEGIN
        (DECODE effadd GEQ 64 =>
                rsph;
                (ckrda NEXT
                virt.real NEXT
                mbr ← MEMH[mar] NEXT
                srctmp ← mbr)
        )
        END;


!Read data from a fullword

rdw:=   BEGIN
        (DECODE effadd GEQ 64 =>
                rspw;
                (ckrda NEXT
                virt.real NEXT
                mbr ← MEMW[mar<0:24>] NEXT
                srctmp ← mbr)
        )
        END;


!read a data byte (we use the halfword routines for the special addresses)

rdbyte:=BEGIN
        (DECODE effadd GEQ 64 =>
         rspb:= BEGIN
                rsph NEXT
                (DECODE byteselect =>
                 \0     srctmp ← srctmp<17:24>;
                 \1     srctmp ← srctmp<25:32>
                )
                END;
         rdb:=  BEGIN
                ckrda NEXT
                virt.real NEXT
                mbr ← MEMB[mar @ byteselect] NEXT
                srctmp ← mbr
                END
        )
        END;

!read an instruction from a word

riw:=   BEGIN
        (DECODE Effadd GEQ 64 =>
                (DECODE effadd<10> =>
                        (rspw NEXT irshd ← srctmp<1:32>);
                        (NF ← 1; irshd ← TRAPREG NEXT BAILOUT (fetch))=
                ));
                (ckria NEXT
                virt.real NEXT
                mbr ← MEMW[mar<0:24>] NEXT
                irshd ← mbr)
        )
        END;
```

```
!Write into a halfword

wrthw:= BEGIN
        (DECODE effadd GEQ 64 =>
                wsphi
                (ckwa NEXT
                virt.real ; mbr ← dsttmp(1:32) NEXT
                MEMH[mar] ← mbr(16:31))
        )
        END;


!Write a full word

wrtw:=  BEGIN
        (DECODE effadd GEQ 64 =>
                wspwi
                (ckwa NEXT
                virt.real ; mbr ← dsttmp(1:32) NEXT
                MEMW[mar(0:24)] ← mbr)
        )
        END;


!Write into a byte

wrtbyte:=BEGIN
        (DECODE effadd GEQ 64 =>
         wspb:= BEGIN
                rsph NEXT
                (DECODE byteselect =>
                 \0     dsttmp ← dsttmp(25:32)@srctmp(25:32);
                 \1     dsttmp ← srctmp(17:24)@dsttmp(25:32)
                ) NEXT
                wsph
                END;
        wrtb:=  BEGIN
                ckwa NEXT
                virt.real NEXT
                mbr ← dsttmp (1:32) NEXT
                MEMB[mar@byteselect] ← mbr(24:31)
                END
        )
        END;
```

```
!Effective address calculation and operand fetch and store routines


!sign extension

signext:= BEGIN
        (DECODE srctmp <17> =>
                srctmp<0:16> ← "00000;
                srctmp<0:16> ← "1FFFF
        )
        END;

!Literal addressing mode  -  mode 0

Literal:=BEGIN
        (DECODE noextend =>
        \0      (srctmp ← PLUS opadd PLEN;
                 pregtmp<1:32> ← PREGW[index] NEXT
                 pregtmp<0> ← pregtmp<1>
                );
        \1      (srctmp ← opadd; pregtmp ← PREGW[index])
        ) NEXT
        (IF index NEQ 0 =>
                (IF noflags =>
                        srctmp ← (srctmp + pregtmp)<32:0> NEXT
                        BAILOUT literal
                ) NEXT
                alu33 ← srctmp<1:32> + pregtmp<1:32> NEXT
                CF ← alu33<0>;  !# carry out of word sign position
                srctmp<1:32> ← alu33<1:32>;
                srctmp<0> ← (srctmp<0> + pregtmp<0> + alu33<0>)<0> NEXT
                (IF cfonly => BAILOUT literal) NEXT
                EF ← (srctmp<0:17> NEQ 0) AND (srctmp<0:17> NEQ #777777);
                OF ← (srctmp<0> NEQ srctmp<1>) NEXT
                (IF OF AND (NOT notrap AND NOT OT) => trapflag ← 1)
        )
        END;

!Direct addressing mode - mode 1


Direct:=BEGIN
        (DECODE index NEQ 0 =>
                effadd ← opadd;
                effadd ← (opadd + PREGW[index])<15:0>
        )
        END;
!Relative addressing mode - mode 2


Relative:=BEGIN
        (IF index EQL 0 => effadd ← (opadd + PC)<15:0>);  !overflow ignored
        (IF index EQL 1 => effadd ← (opadd + PC + PREGW[1])<15:0>);
        (IF index GTR 1 => effadd ← (opadd + PREGW[1] + PREGW[index])<15:0>)
        END;
!Indirect addressing mode - mode 3

Indirect:=BEGIN
        effadd ← opadd NEXT
        rdw NEXT
        effadd ← srctmp<17:32> NEXT
        (IF index NEQ 0 =>
                effadd ← (effadd + PREGW[index])<15:0>)
        END;

!Word operand fetch

wopfetch:= BEGIN
        (DECODE admod =>
                (DECODE NOT evmodz =>
                        (SV ← 1 NEXT pviolation);
                        (Literal)
                );
                (Direct NEXT rdw);
                (Relative NEXT rdw);
                (Indirect NEXT rdw)
        ) NEXT
        (IF NOT noextend AND (admod NEQ 0) => srctmp<0> ← srctmp<1>)
        END;



!halfword operand fetch

hopfetch:=BEGIN
        (DECODE admod =>
                (DECODE NOT e-modz =>
                        (SV ← 1 NEXT pviolation);
                        (literal)
                );
                (direct NEXT rdhw);
                (relative NEXT rdhw);
                (indirect NEXT rdhw)
        ) NEXT
        (IF (NOT noextend) AND (admod NEQ 0) => signext)
        END;

!Byte operand fetch:

bopfetch:=BEGIN
        (DECODE admod=>
                (DECODE NOT exmodz =>
                        (SV ← 1 NEXT pviolation);
                        (literal NEXT
                        (DECODE byteselect =>
                                srctmp ← srctmp<17:24>;
                                srctmp ← srctmp<25:32>
                        ))
                );
                (Direct NEXT rdbyte);
                (relative NEXT rdbyte);
                (indirect NEXT rdbyte)
        )
        END;

!Word operand store

wopstore:=BEGIN
        (DECODE admod =>
                (SV ← 1 NEXT pviolation);
                (direct);
                (relative);
                (indirect)
        )NEXT
        wrtw
        END;


!halfword operand store

hopstore:= BEGIN
        (decode admod =>
                (SV ← 1 NEXT pviolation);
                (direct);
                (relative);
                (indirect)
        )NEXT
        wrthw
        END;


!Byte operand store

bopstore:=BEGIN
        (DECODE admod =>
                (SV ← 1 NEXT pviolation);
                (direct);
                (relative);
                (indirect)
        )NEXT
        wrtbyte
        END;
```

```
!GYK12 Instruction set

!Each instruction is described by a seperate routine below.  Several
!steps were used for each routine:
!         - reset condition codes
!         - set mode flags so utility routines behave
!         - calculate operand address and fetch operand into "srctmp"
!         - perform operation using srctmp and a process register (usually)
!         - store result from dsttmp into destination
!         - set condition codes
!Not all instructions contain all of the above steps; but this should
!serve as a good outline for understanding the isp.
!
!Note that memory access violations, etc. will cause instruction
!operation to be aborted.  (See trap and swaptwo, above)

!Routines for resetting and setting the condition codes:

 resetflags:= (CF ← 0; OF ← 0; EF ← 0; LF ← 0 );

 setef1:= (EF ← ((dsttmp<0:17> NEQ 0) AND (dsttmp<0:17> NEQ "3FFFF)));

 setef2:= (EF ← ((dsttmp<1:17> NEQ 0) AND (dsttmp<1:17> NEQ "1FFFF)));

 setof:= (OF ← (OF OR (dsttmp<0> NEQ dsttmp<1>)));

 setoflogical:= (OF ← OF OR dsttmp<0>));

 setcf:= ( CF ← CF OR dsttmp<0> );

 traptest:=      (IF OF AND NOT OT => trapflag ← 1);
```

```
!Load (Register) Instruction

ldf:=   BEGIN               !load data full
        resetflags;
        cfonly ← 0 NEXT
        wopfetch NEXT
        PREGW[accum] ← srctmp<1:32>
        END;


LDH:=   BEGIN               !Load data halfword
        resetflags;
        cfonly ← 0 NEXT
        hopfetch NEXT
        PREGW[accum] ← srctmp<1:32>
        END;


lmh:=   BEGIN               !load most half
        resetflags;
        pronly ← 1; cfonly ← 0 NEXT
        hopfetch NEXT
        PREGW[accum]<0:15> ← srctmp<17:32>
        END;


ldu:=   BEGIN               !Load from upper byte
        resetflags;
        pronly ← 1; cfonly ← 0; byteselect ← 0 NEXT
        hopfetch NEXT
        PREGW[accum] ← srctmp<25:32>    !note PREGW[accum]<0:23> ← 0
        END;


ldl:=   BEGIN               !Load from lower byte
        resetflags;
        pronly ← 1; cfonly ← 0; byteselect ← 1 NEXT
        hopfetch NEXT
        PREGW[accum] ← srctmp<25:32>    !note PREGW[accum]<0:23> ← 0
        END;


laf:=   BEGIN               !Load absolute value full
        resetflags;
        pronly ← 1 NEXT
        wopfetch NEXT
        (DECODE srctmp<0> =>
        \0        dsttmp ← srctmp;
        \1        dsttmp ← (MINUS srctmp)<32:0>
        ) NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setcf; setofl NEXT
        traptest
        END;


LAH:=   BEGIN               !Load absolute value half
        resetflags;
        pronly ←1 NEXT
        hopfetch NEXT
        (DECODE srctmp<0> =>
                  dsttmp ← srctmp;
                  dsttmp ← (MINUS srctmp)<32:0>
        ) NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setof; setofl NEXT
        traptest
        END;


lcf:=   BEGIN               !Load two's complement full
        resetflags;
        pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← (minus srctmp)<32:0> NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setof; setofl NEXT
        traptest
        END;
```

```
lchw:=  BEGIN   !load two's complement halfword
        resetflags;
        pronly ← 1 NEXT
        bopfetch NEXT
        dsttmp ← (minus acctmp)<32:0> NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setof; setef1 NEXT
        trantest
        END;

!Store (Register) Instruction

sdf:=   BEGIN   !store full word
        resetflags;
        exmodz ←  NEXT
        dsttmp ← PREGW[accum] NEXT
        wopstore;
        setef2
        END;

sdh:=   BEGIN           !store halfword
        resetflags;
        exmodz ← 1 NEXT
        dsttmp ← PREGW[accum] NEXT
        hopstore;
        setef2
        END;


smh:=   BEGIN           !store most significant halfword
        resetflags;
        exmodz ← 1 NEXT
        dsttmp ← PREGW[accum]<0:15> NEXT
        hopstore;
        setef2
        END;

sdu:=   BEGIN           !store into upper byte
        pronly ← 1; exmodz ← 1; byteselect ← 0 NEXT
        dsttmp ← PREGW[accum]<24:31> NEXT
        bopstore
        END;


sdl:=   BEGIN           !store into lower byte
        pronly ← 1; exmodz ← 1; byteselect ← 1 NEXT
        dsttmp ← PREGW[accum]<24:31> NEXT
        bopstore
        END;

!Move Instruction

MZF:=   BEGIN           !Move zeroes, fullword
        exmodz ← 1;
        dsttmp ← 0 NEXT
        wopstore
        END;

MZH:=   BEGIN           !Move zeroes, halfword
        exmodz ← 1;
        dsttmp ← 0 NEXT
        hopstore
        END;

MIU:=   BEGIN           !Move immediate into upper byte
        effadd ← opadd NEXT
        dsttmp ← immed  NEXT
        byteselect ← 0 NEXT
        writbyte
        END;

MIL:=   BEGIN           !move immediate into lower byte
        effadd ← opadd NEXT
        dsttmp ←  immed NEXT
        byteselect ← 1 NEXT
        writbyte
        END;


!Exchange Instruction

EXF:=   BEGIN                   !exchange full
!* this is read-modify-write
        resetflags;
        exmodz ← 1 NEXT
        dsttmp ← PPEGH[accum] NEXT
        setef2;
        hopfetch NEXT
        PPEGH[accum] ← srctmp(1:32);
        wrtw
        END;


EXH:=   BEGIN                   !exchange halfword
!* this is read-modify-write
        resetflags;
        exmodz ← 1; pronly ← 1 NEXT
        dsttmp ← PPEGH[accum] NEXT
        setef2;
        hopfetch NEXT
        PPEGH[accum] ← srctmp(1:32);
        wrthw
        END;
```

```
!Arithmetic instructions

!Add common procedure for ADF, ADH, RAF and RAH

ADD:=   BEGIN
        pregtmp ← PREGW[accum] NEXT pregtmp<0> ← pregtmp<1> NEXT
        dsttmp ← srctmp<1:32> + pregtmp<1:32> NEXT
        CF ← CF OP dsttmp<0> NEXT
        dsttmp<0> ← (dsttmp<0> + srctmp<0> + pregtmp<0>)<0>
        END;


ADF:=   BEGIN              !Add full word
        resetflags;
        pronly ← 1; noflag ← 1 NEXT
        wopfetch NEXT
        add NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setof; setefl NEXT
        traptest
        END;


ADH:=   BEGIN              !add halfword
        resetflags;
        pronly ← 1; noflag ← 1 NEXT
        hopfetch NEXT
        add NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setof; setefl NEXT
        traptest
        END;


ALF:=   BEGIN              !add logical fullword
        resetflags;
        pronly ← 1; noextend ← 1 NEXT
        wopfetch NEXT
        dsttmp ← srctmp<1:32> + PREGW[accum] NEXT
        CF ← CF OP dsttmp<0> NEXT
        dsttmp<0> ← (dsttmp<0> + srctmp<0>)<0> NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setoflogical; setefl
        END;


ALH:=   BEGIN              !add logical halfword
        resetflags;
        pronly ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        dsttmp ← srctmp<1:32> + PREGW[accum] NEXT
        CF ← CF OP dsttmp<0> NEXT
        dsttmp<0> ← (dsttmp<0> + srctmp<0>)<0> NEXT
        PREGW[accum] ← dsttmp<1:32>;
        setoflogical; setefl
        END;


RAF:=   BEGIN              !replace add fullword
        resetflags;
        exmodz ← 1; pronly ← 1 NEXT
        wopfetch NEXT
        add NEXT
        wrtw;
        setof; setefl NEXT
        traptest
        END;


RAH:=   BEGIN              !replace add halfword
        resetflags;
        exmodz ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        add NEXT
        wrthw;
        setof; setefl NEXT
        traptest
        END;


SBF:=   BEGIN              !subtract fullword
        resetflags;
        pronly ← 1; noflag ← 1 NEXT
        wopfetch NEXT
        pregtmp ← PREGW[accum] NEXT pregtmp<0> ← pregtmp<1> NEXT
        dsttmp ← pregtmp<1:32> - srctmp<1:32> NEXT
```

```
          CF ← CF OR NOT dsttmp<0> NEXT
          dsttmp<0> ← (pregtmp<0> - srctmp<0> - dsttmp<0>)<0> NEXT
          PREGW[accum] ← dsttmp<1:32>;
          setof; setefl NEXT
          traptest
          END;


SBH:=     BEGIN              !subtract halfword
          resetflags;
          pronly ← 1; noflag ← 1 NEXT
          hopfetch NEXT
          pregtmp ← PREGW[accum] NEXT pregtmp<0> ← pregtmp<1> NEXT
          dsttmp ← pregtmp<1:32> - srctmp<1:32> NEXT
          CF ← CF OR NOT dsttmp<0> NEXT
          dsttmp<0> ← (pregtmp<0> - srctmp<0> - dsttmp<0>)<0> NEXT
          PREGW[accum] ← dsttmp<1:32>;
          setof; setefl NEXT
          traptest
          END;


SLF:=     BEGIN              !subtract logical full
          resetflags;
          pronly ← 1; noextend ← 1 NEXT
          wopfetch NEXT
          dsttmp ← PREGW[accum] - srctmp<1:32> NEXT
          CF ← CF OR NOT dsttmp<0> NEXT
          dsttmp<0> ← (- srctmp<0> - dsttmp<0>)<0> NEXT
          PREGW[accum] ← dsttmp<1:32>;
          setoflogical; setefl
          END;


SLH:=     BEGIN              !Subtract logical halfword
          resetflags;
          pronly ← 1; noextend ← 1 NEXT
          hopfetch NEXT
          dsttmp ← PREGW[accum] - srctmp<1:32> NEXT
          CF ← CF OR NOT dsttmp<0> NEXT
          dsttmp<0> ← (- srctmp<0> - dsttmp<0>)<0> NEXT
          PREGW[accum] ← dsttmp<1:32>;
          setoflogical; setefl
          END;


RSF:=     BEGIN              !replace subtract fullword
          resetflags;
          exmodz ← 1; pronly ← 1 NEXT
          wopfetch NEXT
          pregtmp ← PREGW[accum] NEXT pregtmp<0> ← pregtmp<1> NEXT
          dsttmp ← srctmp<1:32> - pregtmp<1:32> NEXT
          CF ← CF OR NOT dsttmp<0> NEXT
          dsttmp<0> ← (srctmp<0> - pregtmp<0> - dsttmp<0>)<0> NEXT
          wrtw;
          setof; setefl NEXT
          traptest
          END;


RSH:=     BEGIN              !replace subtract halfword
          resetflags;
          exmodz ← 1; pronly ← 1 NEXT
          hopfetch NEXT
          pregtmp ← PREGW[accum] NEXT pregtmp<0> ← pregtmp<1> NEXT
          dsttmp ← srctmp<1:32> - pregtmp<1:32> NEXT
          CF ← CF OR NOT dsttmp<0> NEXT
          dsttmp<0> ← (srctmp<0> - pregtmp<0> - dsttmp<0>)<0> NEXT
          wrthw;
          setof; setefl NEXT
          traptest
          END;
```

```
!Multiply and other complex arithmetic instructions
!
!The condition code setting lines have been extracted from the
!following routines and implemented as separate routines so that
!they can be "renamed" during the simulation.  The code settings
!routines preceed the instruction they refer to.

setcef:=BEGIN
        CF ← ((tmp66<65:31> NEQ 0) AND (tmp66<65:31> NEQ "7FFFFFFF) ;
        EF ← ((tmp66<65:15> NEQ 0) AND (tmp66<65:15> NEQ ("7FFFF @ "FFFFFFFF))
        END;


MPF:=   BEGIN            !Multiply fullword
        resetflags;
        pronly ← 1; noflag ← 1 NEXT
        wopfetch NEXT
        OF ← (srctmp<0> NEQ srctmp<1>) NEXT
        traptest NEXT
        (IF OF => BAILOUT mpf) NEXT
        pregtmp ← PREGW[accum] NEXT
        t1 ← (srctmp<1> XOR pregtmp<1>) NEXT
        (IF srctmp<1> => srctmp ← MINUS srctmp<1:32>);
        (IF pregtmp<1> => pregtmp ← MINUS pregtmp<1:32>) NEXT
        tmp66 ← srctmp * pregtmp NEXT
        (IF t1 => tmp66 ← (MINUS tmp66)<65:0>) NEXT
        PREGW[accum<0:2> @ '0] ← tmp66<63:32>;
        PREGW[accum<0:2> @ '1] ← tmp66<31:0>;
        setcef
        END;

MPH:=   BEGIN            !Multiply halfword
        resetflags;
        pronly ← 1; noflag ← 1 NEXT
        hopfetch NEXT
        OF ← (srctmp<0> NEQ srctmp<1>) NEXT
        traptest NEXT
        (IF OF => BAILOUT mph) NEXT
        pregtmp ← PREGW[accum] NEXT
        t1 ← (srctmp<1> XOR pregtmp<1>) NEXT
        (IF srctmp<1> => srctmp ← MINUS srctmp<1:32>);
        (IF pregtmp<1> => pregtmp ← MINUS pregtmp<1:32>) NEXT
        tmp66 ← srctmp * pregtmp NEXT
        (IF t1 => tmp66 ← (MINUS tmp66)<65:0>) NEXT
        PREGW[accum] ← tmp66<31:0>;
        setcef
        END;

setdef:= (EF ← ((PREGW[(t4 + 1)<3:0>]<0:16> NEQ 0) AND
                (PREGW[(t4 + 1)<3:0>]<0:16> NEQ "1FFFF)));

DIF:=   BEGIN            !Divide fullword
        resetflags;
        pronly ← 1;
        t4 ← accum NEXT
        wopfetch NEXT
        (IF srctmp EQL 0 => OF ← 1; trapflag ← NOT OT) NEXT
        (IF OF => BAILOUT dif) NEXT
        (DECODE t4<3> =>
        \even   t64 ← PREGW[t4] @ PREGW[(t4 + 1)<3:0>];
        \odd    (t64 ← PREGW[t4] NEXT
                 (IF t64<31> => t64<63:32> ← "FFFFFFFF))
        ) NEXT
        t4<3> ← 0 NEXT
        t1 ← (t64<63> XOR srctmp<1>) NEXT
        t1a ← t64<63> NEXT
        (IF t64<63> => t64 ← (MINUS t64)<63:0>);
        (IF srctmp<1> => srctmp ← (MINUS srctmp)<31:0>) NEXT
        (IF ((t64 / srctmp)<63:31> NEQ 0 =>
                OF ← 1; trapflag ← NOT OT NEXT BAILOUT dif) NEXT
        PREGW[(t4 + 1)<3:0>] ← (t64 / srctmp) <31:0> NEXT
        PREGW[t4] ← ((t64 - (PREGW[(t4 + 1)<3:0>] * srctmp))<31:0> NEXT
        (IF t1 => PREGW[(t4+1)<3:0>] ← (MINUS PREGW[(t4 + 1)<3:0>])<31:0>));
        (IF t1a => PREGW[t4] ← (MINUS PREGW[t4])<31:0>) NEXT
        setdef
        END;

DIH:=   BEGIN            !Divide halfword
        resetflags;
        pronly ← 1;
        t4 ← accum NEXT
        hopfetch NEXT
        (IF srctmp EQL 0 => OF ← 1; trapflag ← NOT OT) NEXT
        (IF OF => BAILOUT dih) NEXT
        t1 ← srctmp<1> XOR PREGW[accum]<0> NEXT
        t1a ← PPEGW[accum]<0> NEXT
        (IF srctmp<1> =) srctmp ← (MINUS srctmp)<31:0>);
        (DECODE t1a =>
        \0      t64 ← PPEGW[accum];
        \1      t64 ← MINUS PPEGW[accum]
        ) NEXT
        t4<3> ← 0 NEXT
!=
        (IF (t64 / srctmp)<63:31> NEQ 0 =>
                OF ← 1; trapflag ← NOT DT NEXT BAILOUT dih) NEXT
        PPEGW[(t4 + 1)<3:0>] ← (t64 / srctmp)<31:0> NEXT
        PPEGW[t4] ← (t64 - (PPEGW[(t4 + 1)<3:0>] * srctmp))<31:0> NEXT
        (IF t1 =) PPEGW[(t4 + 1)<3:0>] ← (MINUS PPEGW[(t4 + 1)<3:0>])<31:0>);
        (IF t1a =) PPEGW[t4] ← (MINUS PPEGW[t4])<31:0>) NEXT
        setdef
        END;


RQF :=  BEGIN               !replace square root full (behaves like sdf)
        resetflags;         !SQRT calculation not implemented.
                            !but zero operand is detected.
        exmodz ← 1; pronly ← 1 NEXT
        (IF PPEGW[accum]<0> =) OF ← 1 NEXT traptest NEXT BAILOUT rqf) NEXT
!       dsttmp ← SQRT(PPEGW[accum])
        dsttmp ← PPEGW[accum] NEXT
        uopstore NEXT
        setef2
        END;
```

```
!Transfer (branch) instructions

!Construct transfer address uses same addressing modes as operand fetch
!but direct behaves like an indirect, and indirect behaves like two
!levels of indirection.

XFERFETCH:=
        BEGIN
        pronly ← 1; noextend ← 1; noflags ← 1 NEXT
        (DECODE admod =>
        \0      Literal;
        \1      (Direct NEXT rdhw);       !CF. sec. 5-4c is wrong
        \2      (Relative NEXT srctmp ← effadd);
        \3      (Indirect NEXT rdhw)
        ) NEXT
        srctmp<32> ← 0 NEXT       !PC<15> is always ZERO
        !The following statement is included to prevent obvious infinite
        !  loop from wasting cpu time.  (not part of architecture)
        (IF srctmp<16:31> EQL (PC-2)<15:1> => STOP)
        END;

XFR:=   BEGIN              !Unconditional branch
        xferfetch NEXT
        PC ← srctmp<17:32>
        END;


XLK:=   BEGIN              !Transfer and link ( save address in accum)
!= what if link reg is PC
        xferfetch NEXT
        PREGW[accum]<16:31> ← PC NEXT
        PC ← srctmp<17:32>
        END;


XIN:=   BEGIN              !Transfer on indicators
        (IF (accum AND flags) NEQ 0 =>
                xferfetch NEXT
                PC ← srctmp<17:32>
        )
        END;

XSW:=   BEGIN              !Transfer on test switches
        (IF (accum AND CTS) NEQ 0 =>
                xferfetch NEXT
                PC ← srctmp<17:32>
        )
        END;


XEX:=   BEGIN              !Execute
        xferfetch NEXT
        effadd ← srctmp<17:32> NEXT
        riw NEXT
        execute ← 1        !This flag causes iexec to loop back and repeat.
                           !Note that this locks out interrupts
        END;


!Index Test Instructions
!       If accum = index the value of accumulator before modification
!       is used for indexing

XDO:=   BEGIN              !If accum is nonzero, subtract one and branch
        (IF PREGW[accum]<16:31> NEQ 0 =>
                xferfetch NEXT
                PREGW[accum]<16:31> ← (PREGW[accum]<16:31> - 1)<15:0> NEXT
                PC ← srctmp<17:32>
        )
        END;


XDT:=   BEGIN              !If accum is nonzero, subtract two and branch
        (IF PREGW[accum]<16:30> NEQ 0 =>         !PREGW gtr 1
                xferfetch NEXT
                PREGW[accum]<16:31> ← (PREGW[accum]<16:31> - 2)<15:0> NEXT
                PC ← srctmp<17:32>
        )
        END;


XIO:=   BEGIN              !If accum is nonzero, add one and branch
        (IF PREGW[accum]<16:31> NEQ 0 =>
                xferfetch NEXT
                PREGW[accum]<16:31> ← (PREGW[accum]<16:31> + 1)<15:0> NEXT
                PC ← srctmp<17:32>
        )
        END;


XIT:=   BEGIN               !If accum is nonzero, add two and branch
        (IF PPEGW[accum]<16:30> NEQ 0 =>          !PPEGW gtr 1
                xferfetch NEXT
                PPEGW[accum]<16:31> ← (PPEGW[accum]<16:31> + 2)<15:0> NEXT
                PC ← srctmp<17:32>
        )
        END;


!Process Register Test Instruction


XEF:=   BEGIN               !transfer on zero accumulator
        (IF PREGW[accum] EQL 0 =>
                xferfetch NEXT
                PC ← srctmp<17:32>
        )
        END;


XUF:=   BEGIN               !Transfer on nonzero accumulator
        (IF PPEGW[accum] NEQ 0 =>
                xferfetch NEXT
                PC ← srctmp<17:32>
        )
        END;


XPF:=   BEGIN               !transfer on positive accumulator
        (IF NOT PREGW[accum]<0>  =>
                xferfetch NEXT
                PC ← srctmp<17:32>
        )
        END;


XNF:=   BEGIN               !transfer on negative accumulator
        (IF PREGW[accum]<0>  =>
                xferfetch NEXT
                PC ← srctmp<17:32>
        )
        END;
```

```
!Shift instructions

!Macros to define the shift operand field=

MACRO tally:=   srctmp<17:20> $ !selects a register to contain a shift count

MACRO option:=  srctmp<21:26> $ !selects a shift option (left, right, etc.)

MACRO shifts:=  srctmp<27:32> $ !specifies number of shifts

MACRO getdreg:= pregd ← PREGW[accum] @ PREGW[accum<0:2> @ '1]$

MACRO putdreg:= PREGW[accum] ← pregd<0:31> NEXT
                PREGW[accum<0:2> @ '1] ← pregd<32:63>$


SHF:=    BEGIN              !full word and double word shifts
!        In double word shift:
!        if H is even, PREGW[accum] @ PREGW[accum+1] is used.
!        if H is odd , PREGW[accum] @ PREGW[accum] is used.
         resetflags;
         pronly ← 1; noextend ← 1; cfonly ← 0; nitrap ← 1 NEXT
         Ibopfetch NEXT
         (DECODE option =>
         \00     SARF:=  BEGIN
                         (DECODE PREGW[accum]<0> =>
                         \0      PREGW[accum] ← PREGW[accum] ↑SR0 shifts;
                         \1      PREGW[accum] ← PREGW[accum] ↑SR1 shifts
                         )
                         END;
         \01     unspec;
         \02     SALF:=  BEGIN           !arithmetic left shift
                         PREGW[accum]<1:31> ← PREGW[accum]<1:31> ↑SL0 shifts
                         END;
         \03     unspec;
         \04     SLRF:=  BEGIN           !Logical right shift
                         PREGW[accum] ← PREGW[accum] ↑SR0 shifts
                         END;
         \05     SCRF:=  BEGIN           !Circular right shift
                         PREGW[accum] ← PREGW[accum] ↑RR shifts
                         END;
         \06     SLLF:=  BEGIN           !Logical left shift
                         PREGW[accum] ← PREGW[accum] ↑SL0 shifts
                         END;
         \07     SCLF:=  BEGIN           !Circular left shift
                         PREGW[accum] ← PREGW[accum] ↑RL shifts
                         END;
         \08     SARD:=  BEGIN           !Arithmetic right double
                         getdreg NEXT
                         (DECODE pregd<0> =>
                         \0  pregd ← pregd ↑SR0 shifts;
                         \1  pregd ← pregd ↑SR1 shifts
                         ) NEXT
                         putdreg
                         END;
         \09     unspec;
         \0A     SALD:=  BEGIN           !Arithmetic left double
                         getdreg NEXT
                         pregd<1:63> ← pregd<1:63  ↑SL0 shifts NEXT
                         putdreg
                         END;
         \0B     unspec;
         \0C     SLRD:=  BEGIN           !Logical right double
                         getdreg NEXT
                         pregd ← pregd ↑SR0 shifts NEXT
                         putdreg
                         END;
         \0D     SCRD:=  BEGIN           !circular right double
                         getdreg NEXT
                         pregd ← pregd ↑RR shifts NEXT
                         putdreg
                         END;
         \0E     SLLD:=  BEGIN           !logical left double
                         getdreg NEXT
                         pregd ← pregd ↑SL0 shifts NEXT
                         putdreg
                         END;
         \0F     SCLD:=  BEGIN           !circular left double
                         getdreg NEXT
                         pregd ← pregd ↑RL  shifts NEXT
                         putdreg
                         END;
         \10.11  unspec;unspec;
         \12     SNF:=   BEGIN           !normalize full
                         PREGW[tally] ← shifts NEXT
                     snfloop:=BEGIN
```

```
                        (IF (PPEGH[accum]<0> NEQ PPEGH[accum]<1>)
                         OR (PPEGH[tally] EQL 0) =>
                            BAILOUT shf) NEXT
                        PPEGH[accum]<1:31> ← PPEGH[accum]<1:31> †SL0 1 ;
                        PPEGH[tally] ← (PPEGH[tally] - 1)<31:0> NEXT
                        nofloop
                        END
                    END;
    \13.15  unspec;unspec;unspec;
    \16     SCF:=   BEGIN               !shift and count ones
                    scfloop:=BEGIN
                            (IF shifts EQL 0 => BAILOUT shf) NEXT
                            (IF PPEGH[accum]<0> =>
                                    PPEGH[tally] ← (PPEGH[tally] + 1)<31:0>);
                            shifts ← (shifts - 1)<4:0> NEXT
                            PPEGH[accum] ← PPEGH[accum] †SL0 1 NEXT
                            scfloop
                            END
                    END;
    \17     SCCF:=  BEGIN               !shift circular and count ones
                    sccfloop:=BEGIN
                            (IF shifts EQL 0 => BAILOUT shf) NEXT
                            (IF PPEGH[accum]<0> =>
                                    PPEGH[tally] ← (PPEGH[tally] + 1)<31:0>);
                            shifts ← (shifts - 1)<4:0> NEXT
                            PPEGH[accum] ← PPEGH[accum] †RL 1 NEXT
                            sccfloop
                            END
                    END;
    \18.19  unspec;unspec;
    \1A     SND:=   BEGIN               !normalize double
                    getdreg NEXT
                    PPEGH[tally] ← shifts NEXT
            sndloop:=
                    (IF (pregd<0> EQL pregd<1>) AND (PPEGH[tally] NEQ 0) =>
                        pregd<1:63> ← pregd<1:63> †SL0 1 NEXT
                        PPEGH[tally] ← (PPEGH[tally] - 1)<31:0> NEXT
                        sndloop
                    ) NEXT
                    putdreg
                    END;
    \1B.1E  unspec;unspec;unspec;unspec;
    \1F     RFT:=   BEGIN               !reflect
                    getdreg NEXT
            rftloop:=
                    (IF shifts NEQ 0 =>
                        t1 ← pregd<31>; t1a ← pregd<32> NEXT
                        pregd<0:31> ← pregd<0:31> †SR t1a;
                        pregd<32:63> ← pregd<32:63> †SL t1;
                        shifts ← (shifts - 1)<3:0> NEXT
                        rftloop
                    ) NEXT
                    putdreg
                    END
    )
    END;    !of shf instruction


SHH:=   BEGIN               !shift halfword
    resetflags;
    pronly ← 1; noextend ← 1; cfonly ← 0; notrap ← 1 NEXT
    hopfetch NEXT
    (DECODE option =>
    \00     SARH:=  BEGIN               !arithmetic right
                    (DECODE PPEGH[accum]<16> =>
                    \0  PPEGH[accum]<16:31> ← PPEGH[accum]<16:31> †SR0 shifts;
                    \1  PPEGH[accum]<16:31> ← PPEGH[accum]<16:31> †SR1 shifts
                    )
                    END;
    \01     unspec;
    \02     SALH:=  BEGIN               !arithmetic left
                    PPEGH[accum]<17:31> ← PPEGH[accum]<17:31> †SL0 shifts
                    END;
    \03     unspec;
    \04     SLRH:=  BEGIN               !logical right
                    PPEGH[accum]<16:31> ← PPEGH[accum]<16:31> †SR0 shifts
                    END;
    \05     SCRH:=  BEGIN               !circular right
                    PPEGH[accum]<16:31> ← PPEGH[accum]<16:31> †RR shifts
                    END;
    \06     SLLH:=  BEGIN               !logical left
                    PPEGH[accum]<16:31> ← PPEGH[accum]<16:31> †SL0 shifts
                    END;
    \07     SCLH:=  BEGIN               !circular left
                    PPEGH[accum]<16:31> ← PPEGH[accum]<16:31> †RL shifts
                    END;
```

```
\0B.11   unspec;unspec;unspec;unspec;unspec;unspec;unspec;unspec;unspec;unspec;
\12      SNH:=    BEGIN               !normalize half
                  PPEGW[tally] ← shifts NEXT
                  snhloop:=BEGIN
                           (IF (PPEGW[accum]<16> NEQ PPEGW[accum]<17>)
                            OR (PPEGW[tally] EQL 0) =>
                                  BAILOUT shh) NEXT
                           PPEGW[accum]<16:31> ← PPEGW[accum]<16:31> ↑SL0 1;
                           PPEGW[tally] ← (PPEGW[tally] - 1)<31:0> NEXT
                           snhloop
                           END
                  END;
\13.15   unspec;unspec;unspec;
\16      SCH:=    BEGIN               !shift and count half
              schloop:=BEGIN
                       (IF shifts EQL 0 => BAILOUT shh) NEXT
                       (IF PPEGW[accum]<16> =>
                           PPEGW[tally] ← (PPEGW[tally] + 1)<31:0>) NEXT
                       shifts ← (shifts - 1)<4:0> NEXT
                       PPEGW[accum]<16:31> ← PPEGW[accum]<16:31> ↑SL0 1 NEXT
                       schloop
                       END
                  END;
\17      SCCH:=   BEGIN               !shift circular and count half
              scchloop:=BEGIN
                       (IF shifts EQL 0 => BAILOUT shh) NEXT
                       (IF PPEGW[accum]<16> =>
                           PPEGW[tally] ← (PPEGW[tally] + 1)<31:0>);
                       shifts ← (shifts - 1)<4:0> NEXT
                       PPEGW[accum]<16:31> ← PPEGW[accum]<16:31> ↑RL 1 NEXT
                       scchloop
                       END
                  END;
\18.1F   unspec;unspec;unspec;unspec;unspec;unspec;unspec;unspec
)
END;     !of shh instruction
```

```
!Compare Instructions;

!These can be grouped for the purpose of the ? and M measure
!simulation.  All countable activity takes place in called routines.


CHF :=  BEGIN               !Compare algebraic full
        reset(flags)
        pronly ← 1 NEXT
        wopfetch NEXT
        tmp33 ← (srctmp MINUS PREGW[accum])<32:0> NEXT
        EF ← (tmp33 EQL 0) NEXT
        OF ← NOT EF AND NOT tmp33<32>;
        LF ← tmp33<32>
        END;


CMH :=  BEGIN               !Compare algebraic half
        reset(flags)
        pronly ← 1 NEXT
        hopfetch NEXT
        tmp33 ← (srctmp MINUS PREGW[accum])<32:0> NEXT
        EF ← (tmp33 EQL 0) NEXT
        OF ← NOT EF AND NOT tmp33<32>;
        LF ← tmp33<32>
        END;


CLU :=  BEGIN               !compare logical upper byte
        reset(flags)
        pronly ← 1; noextend ← 1 NEXT
        byteselect ← 0 NEXT
        bopfetch NEXT
        EF ← (srctmp EQL PREGW[accum]<24:31>);
        OF ← (srctmp GTR PREGW[accum]<24:31>);
        LF ← (srctmp LSS PREGW[accum]<24:31>)
        END;


CLL :=  BEGIN               !compare logical lower byte
        reset(flags)
        pronly ← 1; noextend ← 1 NEXT
        byteselect ← 1 NEXT
        bopfetch NEXT
        EF ← (srctmp EQL PREGW[accum]<24:31>);
        OF ← (srctmp GTR PREGW[accum]<24:31>);
        LF ← (srctmp LSS PREGW[accum]<24:31>)
        END;


CLF :=  BEGIN               !compare logical full
        reset(flags)
        pronly ← 1; noextend ← 1 NEXT
        wopfetch NEXT
        EF ← (srctmp<1:32> EQL PREGW[accum]);
        OF ← (srctmp<1:32> GTR PREGW[accum]);
        LF ← (srctmp<1:32> LSS PREGW[accum] )
        END;


CLH :=  BEGIN               !compare logical halfword
        reset(flags)
        pronly ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        EF ← (srctmp<17:32  EQL PREGW[accum]<16:31>);
        OF ← (srctmp<17:32> GTR PREGW[accum]<16:31>);
        LF ← (srctmp<17:32> LSS PREGW[accum]<16:31>)
        END;


CSF :=  BEGIN               !Compare masked full
        reset(flags)
        pronly ← 1; noextend ← 1 NEXT
        wopfetch NEXT
        EF ← ((srctmp<1:32> AND MASKREG) EQL (PREGW[accum] AND MASKREG));
        OF ← ((srctmp<1:32> AND MASKREG) GTR (PREGW[accum] AND MASKREG));
        LF ← ((srctmp<1:32> AND MASKREG) LSS (PREGW[accum] AND MASKREG))
        END;


CSH :=  BEGIN               !Compare masked half
        reset(flags)
        pronly ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        EF ← ((srctmp<17:32> AND MASKREG<16:31>) EQL
              (PREGW[accum]<16:31> AND MASKREG<16:31>));
        OF ← ((srctmp<12:32> AND MASKREG<16:31>) GTR
                (PPREGH[accum]<16:31> AND MASKREG<16:31>));
        LF ← ((srctmp<12:32> AND MASKREG<16:31>) LSS
                (PPREGH[accum]<16:31> AND MASKREG<16:31>));
        END;


CGF:=   BEGIN               !Compare gate, full
        resetflags;
        pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← (srctmp MINUS PPREGH[accum])<32:0> NEXT
        (DECODE (dsttmp<0> NEQ dsttmp<1>) OR MASKREG<0> =>
        \0      BEGIN
                (IF dsttmp<1> => dsttmp ← (MINUS dsttmp)<32:0>) NEXT
                EF ← (dsttmp<1:32> EQL MASKREG);
                OF ← (dsttmp<1:32> GTR MASKREG);
                LF ← (dsttmp<1:32> LSS MASKREG);
                END;
        \1      OF ← 1
        )
        END;


CGH:=   BEGIN               !Compare gated half
        resetflags;
        pronly ← 1 NEXT
        hopfetch NEXT
        dsttmp ← (srctmp MINUS PPREGH[accum])<32:0> NEXT
        (DECODE (dsttmp<0> NEQ dsttmp<1>) OR MASKREG<0> =>
        \0      BEGIN
                (IF dsttmp<1> => dsttmp ← (MINUS dsttmp)<32:0>) NEXT
                EF ← (dsttmp<1:32> EQL MASKREG);
                OF ← (dsttmp<1:32> GTR MASKREG);
                LF ← (dsttmp<1:32> LSS MASKREG)
                END;
        \1      OF ← 1
        )
        END;


MTH:=   BEGIN               !Modify and test half
        resetflags;
        rxmodz ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        t32 ← accum NEXT
        (IF accum<0> => t32<0:27> ← "FFFFFFF) NEXT
        dsttmp ← (srctmp + t32)<32:0> NEXT
        writhw;
        EF ← (dsttmp<16:32> EQL 0) NEXT
        OF ← (NOT EF AND NOT dsttmp<16>) NEXT
        LF ← dsttmp<16> NEXT
        CF ← ((dsttmp<0:17> NEQ "1FFFF) AND (dsttmp<0:17> NEQ 0)) NEXT
        (IF CF AND NOT OT => trapflag ← 1)
        END;
```

!Logic Instructions

```
IOF :=  BEGIN              !Inclusive or full
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        wopfetch NEXT
        PREGW[accum] ← srctmp<1:32> OR PREGW[accum]
        END;


IOH :=  BEGIN              !Inclusive or half
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        PREGW[accum]<16:31> ← srctmp<17:32> OR PREGW[accum]<16:31>
        END;


RIF :=  BEGIN              !replace inclusive or full
        exmodz ← 1; pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← PREGW[accum] OR srctmp<1:32> NEXT
        wrtw
        END;

RIH :=  BEGIN              !replace inclusive or half
        exmodz ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        dsttmp ← PREGW[accum]<16:31> OR srctmp<17:32> NEXT
        wrthw
        END;

EOF :=  BEGIN              !Exclusive or full
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        wopfetch NEXT
        PREGW[accum] ← srctmp<1:32> XOR PREGW[accum]
        END;


EOH :=  BEGIN              !Exclusive or half
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        PREGW[accum]<16:31> ← srctmp<17:32> XOR PREGW[accum]<16:31>
        END;


REF :=  BEGIN              !replace exclusive or full
        exmodz ← 1; pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← PREGW[accum] XOR srctmp<1:32> NEXT
        wrtw
        END;

REH :=  BEGIN              !replace exclusive or half
        exmodz ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        dsttmp ← PREGW[accum]<16:31> XOR srctmp<17:32> NEXT
        wrthw
        END;


ANF :=  BEGIN              !And full
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        wopfetch NEXT
        PREGW[accum] ← srctmp<1:32> AND PREGW[accum]
        END;


ANH :=  BEGIN              !And half
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        PREGW[accum]<16:31> ← srctmp<17:32> AND PREGW[accum]<16:31>
        END;


RNF :=  BEGIN              !replace and full
        exmodz ← 1; pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← PREGW[accum] AND srctmp<1:32> NEXT
        wrtw
        END;
```

```
RNH:=   BEGIN               !replace and half
        exmodz ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        dsttmp ← PREGW[accum]<16:31> AND srctmp<17:32> NEXT
        writh
        END;

SSF:=   BEGIN               !selective substitute full
        exmodz ← 1; pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← (srctmp<1:32> AND NOT MASKREG) OR
                 (PREGW[accum] AND MASKREG) NEXT
        writw
        END;
```

```
!Bit manipulation instructions          6-39


SBI:=   BEGIN               !set bit in halfword
        exmodz ← 1 NEXT
        hopfetch NEXT
        dsttmp ← srctmp NEXT
        bitdst[accum] ← 1 NEXT
        wrthw
        END;


RBI:=   BEGIN
        exmodz ← 1 NEXT
        hopfetch NEXT
        dsttmp ← srctmp NEXT
        bitdst[accum] ← 0 NEXT
        wrthw
        END;


TSZ:=   BEGIN               !Test bit and skip if zero
        exmod  ← 1 NEXT
        hopfetch NEXT
        (IF NOT bitsrc[accum] => PC ← (PC + 2)<15:0>)
        END;


TSO:=   BEGIN               !Test bit and skip if one
        exmodz ← 1 NEXT
        hopfetch NEXT
        (IF bitsrc[accum] => PC ← (PC + 2)<15:0>)
        END;


TSI:=   BEGIN               !Test and conditionally insert/skip
        exmodz ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        (IF srctmp EQL 0 => PC ← (PC + 2)<15:0>) NEXT
        dsttmp ← (srctmp<17:32>  OR PREGH[accum]<16:31>) NEXT
        wrthw NEXT
        effadd ← PC NEXT
        riw NEXT
        PC ← (PC + 2)<15:0> NEXT        !execute another instruction
        execute ← 1                     !and disallow interrupts
        END;
```

```
!Format instructions

!Macros to define format operand fields:

MACRO dest:=    srctmp<17:20> $

MACRO opt:=     srctmp<25:26> $

!Macro "shifts" from the shift instructions is also used here


FEF:=   BEGIN              !Format extract full
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        (DECODE opt =>
         \0     dsttmp ← PREGW[accum] !SR0 shifts;
         \1     dsttmp ← PREGW[accum] !RR shifts;
         \2     dsttmp ← PREGW[accum] !SL0 shifts;
         \3     dsttmp ← PREGW[accum] !RL shifts
        ) NEXT
        PREGW[dest] ← (dsttmp<1:32> AND MASKREG)
        END;


FEH:=   BEGIN              !Format extract half
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        (DECODE opt =>
         \0     dsttmp ← PREGW[accum]<16:31> !SR0 shifts;
         \1     dsttmp ← PREGW[accum]<16:31> !RR shifts;
         \2     dsttmp ← PREGW[accum]<16:31> !SL0 shifts;
         \3     dsttmp ← PREGW[accum]<16:31> !RL shifts
        ) NEXT
        PREGW[dest]<16:31> ← (dsttmp<17:32> AND MASKREG<16:31>)
        END;


FIF:=   BEGIN              !Format insert full
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        (DECODE opt =>
         \0     dsttmp ← PREGW[accum] !SR0 shifts;
         \1     dsttmp ← PREGW[accum] !RR shifts;
         \2     dsttmp ← PREGW[accum] !SL0 shifts;
         \3     dsttmp ← PREGW[accum] !RL shifts
        ) NEXT
        PREGW[dest] ← (PREGW[dest] AND NOT MASKREG) OR
                      (dsttmp<1:32> AND MASKREG)
        END;


FIH:=   BEGIN              !Format insert half
        resetflags;
        pronly ← 1; cfonly ← 0; notrap ← 1; noextend ← 1 NEXT
        hopfetch NEXT
        (DECODE opt =>
         \0     dsttmp ← PREGW[accum]<16:31> !SR0 shifts;
         \1     dsttmp ← PREGW[accum]<16:31> !RR shifts;
         \2     dsttmp ← PREGW[accum]<16:31> !SL0 shifts;
         \3     dsttmp ← PREGW[accum]<16:31> !RL shifts
        ) NEXT
        PREGW[dest]<16:31> ← (PREGW[dest]<16:31> AND NOT MASKREG<16:31>)
                     OR (dsttmp<17:32> AND MASKREG<16:31>)
        END;
```

```
!Program level change instructions

TXP:=   BEGIN               !Call executive program level and link
        (IF EE AND (APL EQL 2) => EE ← 0 NEXT BAILOUT (evec) NEXT
        noflags ← 1; prnnly ← 1; noextend ← 1 NEXT
        bopfetch NEXT               !This operand is passed to the new p.l.
        (DECODE LC =>               !LC comes from the indicator register
        \0      savregs;
        \1      savr0
        ) NEXT
        HBASEPAGE[(XPL*"40+"30)<11:0>]<2:7> ← APL;      !pass link operand
        HBASEPAGE[(XPL*"40+"31)<11:0>] ← srctmp<17:32> NEXT
        PLLPEG ← BASEPAGE[((XPL * "20) + "18)<10:0>] NEXT
        PPL ← APL NEXT
        (IF NOT srctmp<17> =>
                (DECODE APL<0:1> =>     !reset status bit for active p.l.
                 \0     PS1[APL<2:5>] ← 0;
                 \1     PS2[APL<2:5>] ← 0;
                 \2     PS3[APL<2:5>] ← 0;
                 \3     PS4[APL<2:5>] ← 0
                )
        ) NEXT
        (DECODE XPL<0:1> =>                 !set the status bit for executive p.l.
         \0     PS1[XPL<2:5>] ← 1;
         \1     PS2[XPL<2:5>] ← 1;
         \2     PS3[XPL<2:5>] ← 1;
         \3     PS4[XPL<2:5>] ← 1
        ) NEXT
        LL1 ← 1; PLLFF ← 1;         !set p.l. lock
        APL ← XPL NEXT              !change p.l. to executive
        (DECODE C =>                !L is in the PLLPEG - load the new registers
         \0     (loadregs; loadpcar ; LOCKPCAR ← 0);
         \1     (loadregs; load4pcar; LOCKPCAR ← 1);
         \2     (loadr0; loadpcar ; LOCKPCAR ← 0);
         \3     (loadr0; load4pcar; LOCKPCAR ← 1)
        )
        END;
TCP:=   BEGIN              !Call program level and link
        (IF EE AND (APL EQL 2) => EE ← 0 NEXT BAILOUT levec) NEXT
        pronly ← 1; noflags ← 1; noextend ← 1 NEXT
        bopfetch NEXT             !This operand gets passed to the new p.l.
        (DECODE LC =>
                saveregs;
                savr0
        ) NEXT
        (DECODE srctmp<18> =>    !2nd bit of operand determines called p.l.
         \0     newpl ← HBASEPAGE[(APL*"40+"30)<11:0>]<10:15>;!CPL in PLLREG
         \1     newpl ← 63
        ) NEXT
                                  !set up PLLREG of called p.l.
        BASEPAGE[((newpl * "20) + "10)<10:0>]<16:31> ← srctmp<17:32>;
        BASEPAGE[((newpl) * "20) + "10)<10:0>]<2:7> ← APL NEXT
        (IF NOT srctmp<17> =>    !MSB of operand determines status of current p.l.
                (DECODE APL<0:1> =>     !reset status bit
                 \0     PS1[APL<2:5>] ← 0;
                 \1     PS2[APL<2:5>] ← 0;
                 \2     PS3[APL<2:5>] ← 0;
                 \3     PS4[APL<2:5>] ← 0
                )
        ) NEXT
        (DECODE newpl<0:1> =>             !set status bit of new p.l.
         \0     PS1[newpl<2:5>] ← 1;
         \1     PS2[newpl<2:5>] ← 1;
         \2     PS3[newpl<2:5>] ← 1;
         \3     PS4[newpl<2:5>] ← 1
        ) NEXT
        (IF srctmp<19> =>         !2nd MSB => set queue reg. of called p.l.
                t32 ← BASEPAGE[((newpl) * "20) + "19)<10:0>] NEXT
                bitt32[srctmp<20:24>] ← 1 NEXT
                BASEPAGE[((newpl) * "20) + "19)<10:0>] ← t32
        ) NEXT
        LLI ← 0; PLLFF ← 0 NEXT !reset the level lock
        auction NEXT              !find a new p.l. by auction (may or
                                  !may not be the called level)
        PPL ← APL NEXT
        APL ← newpl NEXT
        PLLREG ← BASEPAGE[((newpl) * "20) + "10)<10:0>] NEXT
        (DECODE C =>
        \0      (loadregs; loadpcar ; LOCKPCAR ← 0);
        \1      (loadregs; load4pcar; LOCKPCAR ← 1);
        \2      (loadr0; loadpcar ; LOCKPCAR ← 0);
        \3      (loadr0; load4pcar; LOCKPCAR ← 1)
        )
        END;
```

```
TIE:=   BEGIN              !Tie program level and link
        (IF LL AND (APL EQL 2) => LL ← 0 NEXT BAILOUT level) NEXT
        fetchcycle NEXT    !Semiprivileged instruction
        bronly ← 1; noflags ← 1; noextend ← 1 NEXT
        hopfetch NEXT               !link argument
        (DECODE LC =>               !LC is in indicator register
         \0      savregs;
         \1      savr0
        );
        newpl ← HBASEPAGE[(APL*"40+"30)<11:0>]<10:15> NEXT
        HBASEPAGE[(newpl*"40+"30)<11:0>]<2:7> ← APL;
        HBASEPAGE[(newpl*"40+"31)<11:0>] ← srctmp<17:32> NEXT
        PLLREG ← BASEPAGE[((newpl * "20) + "10)<10:0>] NEXT
        PPL ← APL NEXT
        (IF NOT srctmp<17> =>
                (DECODE APL<0:1> =>
                 \0      PS1[APL<2:5>] ← 0;
                 \1      PS2[APL<2:5>] ← 0;
                 \2      PS3[APL<2:5>] ← 0;
                 \3      PS4[APL<2:5>] ← 0
                )
        ) NEXT
        (DECODE newpl<0:1> =>
         \0      PS1[newpl<2:5>] ← 1;
         \1      PS2[newpl<2:5>] ← 1;
         \2      PS3[newpl<2:5>] ← 1;
         \3      PS4[newpl<2:5>] ← 1
        ) NEXT
        APL ← newpl;
        LLI ← 1; PLLFF ← 1 NEXT   !set level lock
        (DECODE C =>
         \0      (loadregs; loadpcar ; LOCKPCAR ← 0);
         \1      (loadregs; load4pcar; LOCKPCAR ← 1);
         \2      (loadr0; loadpcar ; LOCKPCAR ← 0);
         \3      (loadr0; load4pcar; LOCKPCAR ← 1)
        )
        END;
```

```
TOR:=   BEGIN           !Test and conditionally reset/skip
        e-modz ← 1 NEXT
        (DECODE a-mod =>               !compute operand address
                (SV ← 1 NEXT pviolation);       !mode 0 not allowed
                direct;
                relative;
                indirect
        ) NEXT
        (DECODE effadd GTR "3F =>                  !Special addresses
        \LEQ3F  BEGIN
                !test for invalid address
                (IF (effadd GTR "1F) AND (effadd NEQ "32) => set(v) NEXT
                rspw NEXT               !read the operand
                (DECODE srctmp EQL 0 => !nonzero operand -
                \NEQ0   BEGIN           !find the first one, reset it and skip
                        tmpctr ← 0 NEXT
                        t37 ← srctmp<1:32> NEXT
                 tqr11:=(DECODE bitt32[tmpctr<3:7>] =>
                          \0     (tmpctr ← (tmpctr + 1)<7:0> NEXT tqr11);
                          \1     (PREGH[accum] ← tmpctr;
                                  bitt32[tmpctr<3:7>] ← 0 NEXT
                                  dsttmp ← t32 NEXT
                                  wrtw NEXT
                                  PC ← (PC + 2)<15:0>)
                        )      !end tqr11
                        END;    !of nonzero special address operand
                !zero operand - p.l. change
                \EQL0   BEGIN
                        (DECODE APL<0:1> =>     !reset APL status bit
                         \0     PS1[APL<2:5>] ← 0;
                         \1     PS2[APL<2:5>] ← 0;
                         \2     PS3[APL<2:5>] ← 0;
                         \3     PS4[APL<2:5>] ← 0
                        ) NEXT
                        LLI ← 0; PLLFF ← 0 NEXT
                        swap    !find another program level
                        END     !of zero special address operand
                )
                END;    !of special addresses
        \GTR3F  BEGIN   !operand address is larger than "3F
                rdw NEXT
                (IF srctmp NEQ 0 =>
                        BEGIN           !nonzero memory operand
                        tmpctr ← 0;
                        t32 ← srctmp<1:32> NEXT
                        tqr11           !reset the first "one" and skip
                        END             !of nonzero memory operand
                ) !end IF
                !IF we get here, then the operand is from memory and =0,
                !Which causes a noop.
                END
        )
        END;    !of TOR
```

```
!I/O Instructions


MACRO Device:= srctmp<26:32> $

MACRO command:= srctmp<17:24> $


DEV:=   BEGIN              !Device command
        prvchk NEXT
        dt ← 0;
        noextend ← 1; noflags ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        iodevice[device] ← command
        !Timeout in 10 us, dt ← 1, bailout dev
        END;

DEX:=   BEGIN              !Device command and exit
        prvchk NEXT
        dt ← 0;
        noextend ← 1; noflags ← 1; pronly ← 1 NEXT
        hopfetch NEXT
        iodevice[device] ← command NEXT
        !Timeout in 10 us, dt ← 1, bailout dex
        (DECODE APL<0:1> =>      !reset APL status bit
         \0      PS1[APL<2:5>] ← 0;
         \1      PS2[APL<2:5>] ← 0;
         \2      PS3[APL<2:5>] ← 0;
         \3      PS4[APL<2:5>] ← 0
        ) NEXT
        LLI ← 0; PLLFF ← 0 NEXT
        mwmp               !Find a new program level
        END;


ITR:=   BEGIN              !input to register - parity checking not implemented
        prvchk NEXT        !priveleged instruction
        ie ← 0;            !parity error bit
        dt ← 0;            !timeout bit
        pronly ← 1;
        noextend ← 1;
        noflags ← 1 NEXT
        hopfetch NEXT
        tmpctr ← 0 NEXT
        PPREGW[accum] ←0 NEXT
        (DECODE (device NEQ 1) AND (device NEQ 2) =>
             (
             (IF device EQL 1 => PREGW[accum] ← monitor);
             (IF device EQL 2 => PREGW[accum] ← maintreg)
             );
    itrlp:=
             (IF tmpctr LEQ 3 =>
                  PREGW[accum] ← PREGW[accum]<8:31> @ inport[device]<1:8>;
                  tmpctr ← (tmpctr + 1)<7:0> NEXT
                  itrlp
             )
        )
        END;     !of itr instruction


OFR:=   BEGIN              !output from register
        prvchk NEXT        !priveleged instruction
        dt ← 0;
        noextend ← 0; pronly ← 1; noflags ← 1 NEXT
        hopfetch NEXT
        (DECODE device GTR 2 =>
         \0      BEGIN
                 (IF device EQL 1 =>
                         monitor ← monitor OR PREGW[accum]);
                 (IF device EQL 2 =>
                         monitor ← monitor AND NOT PREGW[accum])
                 END;
         \1,     BEGIN
                 ioport[device] ← PPREGW[accum]<0:7> NEXT
                 inport[device] ← PPREGW[accum]<8:15> NEXT
                 ioport[device] ← PPREGW[accum]<16:23> NEXT
                 ioport[device] ← PREGW[accum]<24:31>
                 END
        )        !end of decode
        END;
```

```
!Miscellaneous instructions!


HLT:=   BEGIN               !This is a priveleged instruction in GYK12
        ! prvchk NEXT   !For simulation sake, this is an nonprivelened halt.
        (IF (accum EQL 0) OR ((accum AND cts) NEQ 0) =>
                STOP
        )
        END;

MBA:=   BEGIN               !Memory bank assignment - not implemented
        prvchk
!       not described here
        END;

LDD:=   BEGIN               !load cell destination
        sprivchk NEXT
        pronly ← 1; noflags ← 1 NEXT
        hopfetch NEXT
        MBASEPAGE[(APL+"40+"20)<11:0>]<8:15> ← srctmp<25:32>;
        PLLREG<8:15> ← srctmp<25:32>
        END;


LLO:=   BEGIN               !level lock set
        sprivchk NEXT
        LLI ← 1; PLLFF ← 1
        END;


LLR:=   BEGIN               !Level lock reset
        LLI ← 0; PLLFF ← 0
        END;


DIG:=   BEGIN               !Diagnose - not implemented
        prvchk NEXT
        (DECODE accum<2:3> =>
        \0      PC ← opadd;
        \1      unspec;
        \2      digcode ← opadd<0:15>@accum<0:1>;
        \3      cpulou ← opadd<14:15>
        )
        END;
```

```
!*This page is full of hacks.
!Floating Point Option for GYK-12
!Only fixed point operation is perform
!condition codes are resets, but not set except in FCM(compare)


fadi=   BEGIN               !Floating addition
        resetflags;
        notrap ← 1 ; pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← PPEGW[accum] + srctmp<1:32> NEXT
        ! set conditions codes
        PPEGW[accum] ← dsttmp<1:32>
        END;


fcmi=   BEGIN               !Floating compare
        resetflags;
        notrap ← 1 ; pronly ← 1 NEXT
        wopfetch NEXT
        (DECODE srctmp<1> EQL PPEGW[accum]<0> =>
        \no     (OF ← NOT srctmp<1>;
                LF ← srctmp<1>);
        \yes    (dsttmp ← (srctmp<1:32> - PREGW[accum]) NEXT
                EF ← (dsttmp<1:32> EQL 0) NEXT
                OF ← (NOT EF AND NOT dsttmp<1>) NEXT
                LF ← dsttmp<1>)
        )
        END;


fdvi=   BEGIN               !Floating division
        resetflags;
        notrap ← 1 ; pronly ← 1 NEXT
        wopfetch NEXT
        (IF srctmp<1:32> EQL 0 => BAILOUT fdv) NEXT
        t1 ← PREGW[accum]<0> XOR srctmp<1> NEXT
        (IF srctmp<1> => srctmp ← MINUS srctmp<1:32>) NEXT
        (DECODE PPEGW[accum]<0> =>
        \p      tmp33 ← PPEGW[accum];
        \n      tmp33 ← MINUS PPEGW[accum]
        )NEXT
        dsttmp ← ((tmp33 @ 0<15:0>) / srctmp<1:32>)<32:0> NEXT
        (IF t1 => dsttmp ← (MINUS dsttmp)<31:0>) NEXT
        ! set conditions codes
        PPEGW[accum] ← dsttmp<1:32>
        END;


fmpi=   BEGIN               !Floating multipy
        resetflags;
        notrap ← 1 ; pronly ← 1 NEXT
        wopfetch NEXT
        t1 ← (srctmp<1> XOR PREGW[accum]<0>) NEXT
        (IF srctmp<1> => srctmp ← MINUS srctmp<1:32>) NEXT
        (DECODE PREGW[accum]<0> =>
        \p      tmp33 ← PREGW[accum];
        \n      tmp33 ← MINUS PREGW[accum]
        )NEXT
        tmp66 ← ((srctmp * tmp33) SPR 16)<31:0> NEXT
        (IF t1 => tmp66 ← (MINUS tmp66)<65:0>) NEXT
        ! set conditions codes
        PPEGW[accum] ← tmp66<31:0>
        END;


fsbi=   BEGIN               !Floating subtraction
        resetflags;
        notrap ← 1 ; pronly ← 1 NEXT
        wopfetch NEXT
        dsttmp ← PREGW[accum] - srctmp<1:32> NEXT
        ! set conditions codes
        PPEGW[accum] ← dsttmp<1:32>
        END;
```

```
!!nstruction fetch and execute cycle!

!fetch!=BEGIN
        effadd ← PC NEXT
        riw NEXT
        PC ← (PC + 2)<15:0>
        END;


!instruction execution

!exec!= BEGIN
        pronly ← 0; exmodz ← 0; notrap ← 0;
        noflags ← 0; noextend ← 0; execute ← 0 NEXT
        (DECODE opcode =>
                nop;     ! #00  no operation
                fnd;     ! #01  #01 - #05 is floating point opcode
                fcm;     ! #02  !*
                fdv;     ! #03
                fmp;     ! #04
      .         fsb;     ! #05
                noi;     ! #06  noi is a undefined instruction
                noi;     ! #07
                ADF;     ! #08  add full
                SBF;     ! #09  subtract full
                ALF;     ! #0A  add logical full
                SLF;     ! #0B  subtract logical full
                MPF;     ! #0C  multiply full
                DIF;     ! #0D  divide full
                RAF;     ! #0E  replace add full
                RSF;     ! #0F  replace subtract full
                CMF;     ! #10  compare algebraic full
                CLU;     ! #11  compare logical upper byte
                CLF;     ! #12  compare logical full
                CGF;     ! #13  compare gated full
                CSF;     ! #14  compare selective full
                IOF;     ! #15  inclusive or full
                EOF;     ! #16  exclusive or full
                ANF;     ! #17  logical and full
                FEF;     ! #18  formal extract full
                FIF;     ! #19  formal insert full
                SHF;     ! #1A  shift full (and double)
                RQF;     ! #1B  replace square root??
                SBT;     ! #1C  Set Bit in Halfword
                RIF;     ! #1D  replace inclusive or full
                REF;     ! #1E  replace exclusive or full
                RNF;     ! #1F  replace logical and full
                LDF;     ! #20  load data full word
                LDU;     ! #21  load from upper byte
                LAF;     ! #22  load absolute full
                LCF;     ! #23  load two's complement half
                LMH;     ! #24  load most half
                SDU;     ! #25  store into upper byte
                SDF;     ! #26  store data full
                MZF;     ! #27  Move all zeroes, full
                noi;     ! #28
                noi;     ! #29
                noi;     ! #2A
                noi;     ! #2B
                noi;     ! #2C
                MIU;     ! #2D  move into upper byte
                EXF;     ! #2E  exchange full
                SSF;     ! #2F  Selective substitute full
                XEP;     ! #30  unconditional transfer
                XSW;     ! #31  transfer on test switches
                XZF;     ! #32  transfer on zero accumulator
                XPF;     ! #33  transfer on positive accumulator
                XDO;     ! #34  conditional transfer and decrement by 1
                XIO;     ! #35  conditional transfer and increment by 1
                XEX;     ! #36  execute
                TSZ;     ! #37  test bit and skip on 0
                DEV;     ! #38  Device command (priveleged)
                ITR;     ! #39  Input to register (priveleged)
                HLT; .   ! #3A  conditional halt (unconditional in this isp)
                DIG;     ! #3B  Diagnose (priveleged) - not implemented
                TXP;     ! #3C  Call executive PL and link
                TIF;     ! #3D  Tie PL and link
                noi;     ! #3E
                noi;     ! #3F
                LLR;     ! #40  level lock reset
                noi;     ! #41
                noi;     ! #42
                noi;     ! #43
                noi;     ! #44
                noi;     ! #45
                noi;     ! #46
```

```
          nol;     ! "47
          ADH;     ! "48   add half
          SBH;     ! "49   subtract half
          ALH;     ! "4A   add logical half
          SLH;     ! "4B   subtract logical half
          MPH;     ! "4C   multiply half
          DIH;     ! "4D   divide half
          RAH;     ! "4E   replace add half
          RSH;     ! "4F   replace subtract half
          CMH;     ! "50   compare algebraic half
          CLL;     ! "51   compare logical lower byte
          CLH;     ! "52   compare logical half
          CGH;     ! "53   compare gated half
          CSH;     ! "54   compare selective half
          IOH;     ! "55   inclusive or half
          EOH;     ! "56   exclusive or half
          ANH;     ! "57   logical and half
          FEH;     ! "58   formal extract half
          FIH;     ! "59   formal insert half
          SHH;     ! "5A   shift halfword
          TQP;     ! "5B   test and conditionally reset/skip
          RBT;     ! "5C   reset bit in halfword
          RIH;     ! "5D   replace inclusive or half
          REH;     ! "5E   replace exclusive or half
          RNH;     ! "5F   replace logical and half
          LDH;     ! "60   load data halfword
          LDL;     ! "61   load from lower byte
          LAH;     ! "62   load absolute half
          LCH;     ! "63   load two's complement half
          SMH;     ! "64   store most half
          SDL;     ! "65   store into lower byte
          SDH;     ! "66   store data half
          MZH;     ! "67   Move all zeroes; half
          nol;     ! "68
          nol;     ! "69
          nol;     ! "6A
          nol;     ! "6B
          nol;     ! "6C
          MIL;     ! "6D   move into lower byte
          EXH;     ! "6E   exchange half
          MTH;     ! "6F   modify and test half
          XLK;     ! "70   transfer and link
          XIN;     ! "71   transfer on indicators
          XUF;     ! "72   transfer on non-zero accumulator
          XNF;     ! "73   transfer on negative accumulator
          XDT;     ! "74   conditional transfer and decrement by 2
          XIT;     ! "75   conditional transfer and increment by 2
          TBI;     ! "76   test and conditionally insert/skip
          TBO;     ! "77   test bit and skip on one
          DEX;     ! "78   device command and exit (priveleged)
          OFR;     ! "79   output from register
          MBA;     ! "7A   memory bank assignment (not implemented)
          LLO;     ! "7B   level lock set (semi-priveleged)
          TCP;     ! "7C   call program level and link
          LDD;     ! "7D   load call destination (semipriveleged)
          nol;     ! "7E
          nol      ! "7F
  ) NEXT
  (IF execute => iexec)   !restart iexec if execute instruction
  END;
```

```
![illegible]

[illegible]    BEGIN
        (IF [illegible] NEXT
        (IF INTFLAG ← IOC NEXT
        (IF NOT PLLFF =>
            auction NEXT                        !select new program level
            (DECODE C<0> =>                     !hardware initiated p.l. change
                savregs;
                savr0
            )NEXT
            PPL ← APL NEXT
            APL ← newpl NEXT
            PLLREG ← BASEPAGE[(newpl = "201 + "10)<10:0>)] NEXT
            (DECODE C=>
                (loadregs; loadpcar ; LOCKPCAR ← 0);
                (loadregs; load4pcar; LOCKPCAR ← 1);
                (loadr0; loadpcar  ; LOCKPCAR ← 0);
                (loadr0; load4pcar; LOCKPCAR ← 1)
            )
            ) NEXT
            intflag ← 0
        )
        END;

BREAK:=(nop)!!*break point stop here
                !*remove after debug

Icycle:=BEGIN
        int NEXT
        ifetch NEXT
icycl:= iexec NEXT
        (IF trapflag => !check trap condition
                trahd ← TRAPREG;
                trapflag ← 0 NEXT
                icycl
        )
        END


ERALCED

run:=   BEGIN
        (IF NOT stopbit =>
                (IF PC eql brkpnt => break) next !*remove
                icycle NEXT
                run
        )
        END
) !end of nyk12;
```

4. AN/UYK-19 ISPL Description

```
!       AN/UYK 19       UYK19.ISP       v 1.4


!       This is the ISP for the ROLM 1602 (AN/UYK 19) Computer.
!       By Al Dunlop
!       3/21/77



!Three types of instructions have been left out of this ISPL
!       discription of the ROLM 1602. They are "INPUT-OUTPUT
!       WITH ACCUMULATOR", "INPUT-OUTPUT WITHOUT ACCUMULATOR",
!       and "CODE-77 IO WITH ACCUMULATOR" (Figure 3-2, Page 3-13
!       of the "1602 RUGGED NOVA COMPUTER  OPERATION AND MAINTENANCE",
!       1974).  All the other I/O and interupt instructions have been
!       included in this discription.
!The decoding for the "INPUT-OUTPUT" is a bit messey in that the MDR
!       is tested repeatedly to determine the instruction.


!v 1.2  Single precision floating point instructions implemented
!       as fixed point.  Entire INPUT.OUTPUT section changed from
!       DECODEs to IF statements with BAILOUTs.

!v 1.3  ROLM 1666 Resource Management Unit added; KL30 and WD15 7/13/77
!       part in arithmetic, part in I/O; Instr have separate routines

!V 1.4  INDIRECTION HAS TO BE STARTED BY MDR<5>
!       MSB OF ADDR WILL BE IGNORED IF EM<0>=0.EXCEPT INDIRECTION IS POSSIBLE
!       ALL ADDR REGISTER IS 16 BIT WIDE, MODULE 2↑16.  MSB IS IGNORED
!               IF NECESSARY
!       AUTO INCREMENT AND DECREMENT IS ONLY POSSIBLE INSIDE INDIRECT CHAIN.
!       KL30 8/1/77

!v 1.5  Device I/O and interrupt sequence added. LS20.
!V 1.6  CODE 77 I/O WITH ACC ADDED


NOVA:=
(!START ROLM 1602(AN/UYK-19)
DECLARE

MACRO BEGIN := ( $
MACRO END   := ) $
MACRO INCR.PC:= PC<0:15>←(PC<0:15>+1)<15:0> $

Memory[0:65535]<0:15>;          !Main Memory
PC<0:15>;                       !Program Counter
MDR<0:15>;                      !Memory Data Register
AC[0:3]<0:15>;                  !Accumulator Set
AC0<0:15> := AC[0]<0:15>;       !Accumulator 0
AC1<0:15> := AC[1]<0:15>;       !Accumulator 1
AC2<0:15> := AC[2]<0:15>;       !Accumulator 2
AC3<0:15> := AC[3]<0:15>;       !Accumulator 3
SP<0:15>;                       !STACK POINTER
SL<0:15>;                       !STACK LIMIT
STATUS<0:15>;                   !Status of the computer
        ION<0>:=STATUS<0>;      !Interupts ON
        IBN<0>:=STATUS<1>;      !Branching interupt sequences ON
        OVF<0>:=STATUS<2>;      !OVerFlow bit
        CARRY<0>:=STATUS<3>;    !Carry bit
        EM<0>:=STATUS<4>;       !Expanded Memory
        SXMD<0> :=STATUS<5>;    !EXECUTIVE MODE
CPDATA<0:15>;                   !CONTROL PANEL DATA LIGHTS
INTMSK<0:15>;                   !INTERRUPT MASK
SWITCH<0:15>;                   !CONSOLE SWITCHES

        ♦

TMP.NO.OP<0>;                   !For No-ops
    TMPADR<0:15>;               !TEMPARY ADDRESS REGISTER returns value
                                !from EFFECTIVE.ADDRESS.CALCULATION.
    TMPMDR<0:15>;               !TEMPARY Memory Data REGISTER transfers
                                !value to or from Memory.
    tmp.fctn.opt<0:16>;         !Tempary buffer at input of Shifts
    TMP.SHIFTER<0:16>;          !Tempary buffer at input
                                !of No Load/Load Switch
    TMP0REG<0:15>;              !TEMPARY REGISTER #0
    TMP1REG<0:15>;              !TEMPARY REGISTER #1
    TMPDOUBLEREG<0:32>;         !TEMPARY DOUBLE REGISTER
    TMP0SIGN<0>;                !TEMPARY SIGN HOLDER
    TMP1SIGN<0>;                !TEMPARY SIGN HOLDER
    TMPINDIRECT<0>;             !Save indirect bit  #wd15 7/1/77
TD0<0:31>;
TD1<0:31>;
SIGN<>;
```

```
!
!   DEVICE REGISTERS
!
DEV.INREG<0:15>;                 !SOURCE FOR INPUT
DEV.OUTREG<0:15>;                !DESTINATION FOR OUTPUT
DEV.NUMBER<0:5>;                 !DEVICE NUMBER OF INTERRUPTING DEVICE
DEV.IBIT<0:15>;          !INTERRUPT BIT FOR EACH BIT IN MASK WORD
                                 !SET CAUSES INTERRUPT IF NOT MASKED


!   RESOURCE MANAGEMENT

MSR<0:15>;                       !MAP STATUS REGISTER
        XMDC<>  := MSR<0>;       !   USER MODE (EXEC MODE COMP)
        XEM<>   := MSR<1>;       !   EXEC EXPANDED MEMORY
        UEM<>   := MSR<2>;       !   USER EXPANDED MEMORY
        XMD<>   := MSR<3>;       !   EXEC DATA MAP
        UDM<>   := MSR<4>;       !   USER DATA MAP
        DMA<>   := MSR<5>;       !   DMA MAP
        P<>     := MSR<6>;       !   USER R/W/EXECUTE PAGE PROTECTION
        D<>     := MSR<7>;       !   DEFER(INDIRECT) PROTECTION
        I.O<>   := MSR<8>;       !   I/O PROTECTION
        DP<>    := MSR<9>;       !   DMA PROTECTION
!               := MSR<10:12>;   !   RESERVED
        USER<2:0>:=MSR<13:15>;   !   USER OR LAST ACTIVE USER(2-7)

MVR<0:15>;                       !MAP VIOLATION REGISTER
        DMPE<>  := MVR<0>;       !   DMA PROTECTION ERROR
        EPE<>   := MVR<1>;       !   EXEC PROTECTION ERROR
        RPE<>   := MVR<2>;       !   READ PROTECTION ERROR
        WPE<>   := MVR<3>;       !   WRITE PROTECTION ERROR
        DPE<>   := MVR<4>;       !   DEFER PROTECTION ERROR
        IOPE<>  := MVR<5>;       !   I/O PROTECTION ERROR
        PRPE<>  := MVR<6>;       !   PRIVELEGED INST PROTECTION ERROR
        SCPE<>  := MVR<7>;       ! VIOLATION OCCURED DURING SINGLE CLYLE OP
!               := MVR<8:12>;    !   RESERVED
        VUSER<2:0>:=MVR<13:15>;  !   LAST ACTIVE USER

MAPREG[0:511]<0:15>;             !MEMORY MAP REGISTER

MSI<>;                   !MAP SINGLE INSTRUTION
MSD<>;                   !MAP SINGLE DATA
DATREF<>;                !DATA REFERENCE
TEMP1<15:0>;
TEMP2<15:0>;
TEMP3<15:0>;
NMAP<2:0>;
PHYADR<0:19>;
TRAP.INDEX<3:0>;
VIRT.REAL:=     !VIRTUAL TO REAL ADDRESS TRANSLATION
        BEGIN
        (DECODE XMDC =>
                NMAP<2:0> ← 1;          !EXECUTIVE MAP
                NMAP<2:0> ← USER<2:0>   !USER MAP
        ) NEXT
        (IF (MSI OR MSD) AND DATREF =>
                NMAP<2:0> ← USER<2:0> NEXT
                MSD ← 0 NEXT MSI ← 0
        ) NEXT
        (DECODE EM<0> =>
        \0      PHYADR<0:19> ← MAPREG[(NMAP<2:0>*64+TMPADR<1:5>)<8:0>]<6:15>
                                @ TMPADR<6:15>;
        \1      PHYADR<0:19> ← MAPREG[(NMAP<2:0>*64+TMPADR<0:5>)<8:0>]<6:15>
                                @ TMPADR<6:15>
        )
        END;


READ.Memory:=   ! Fill TMPMDR with data in Memory location TMPADR.
        BEGIN
        VIRT.REAL NEXT
        TMPMDR<0:15> ← MEMORY[PHYADR<0:19>]<0:15>
        END;


WRITE.Memory:=  ! Store TMPMDR into Memory location TMPADR.
        BEGIN
        VIRT.REAL NEXT
        MEMORY[PHYADR<0:19>]<0:15> ← TMPMDR<0:15>
        END;

NOP:=   BEGIN
        TMP.NO.OP ← TMP.NO.OP
        END;
```

```
ILLEGAL:=
        BEGIN
        STOP    !* Illegal instructions should be trapped
        END;

R.DATA:=
        BEGIN
        DATREF ← 1 NEXT
        READ.MEMORY NEXT
        DATREF ← 0
        END;

W.DATA:=
        BEGIN
        DATREF ← 1 NEXT
        WRITE.MEMORY NEXT
        DATREF ← 0
        END;


!       I N C R . D E C R . M e m o r y
!       I N C R . D E C R . M e m o r y
!       I N C R . D E C R . M e m o r y


INCR.DECR.Memory:=       !>>>Do the AUTOINCREMENT or AUTODECREMENT
                         !of the special memory locations.
(!START INCR.DECR.Memory


    (DECODE TMPADR<12> =>       !Decrement(=0) or Increment(=1)
        (TMPMDR<0:15>←(TMPMDR<0:15> + 1)<15:0>);
        (TMPMDR<0:15>←(TMPMDR<0:15> - 1)<15:0>)
    )NEXT !END DECODE TMPADR<12>
    (WRITE.Memory)

);!END INCR.DECR.Memory
!A D R . F E T C H  Uses Variables TMPMDR<0:15>, TMPADR<0:15>, EM<0>.
!A D R . F E T C H  Uses Routines READ.Memory, INCR.DECR.Memory.
!A D R . F E T C H


ADR.FETCH:=     !>>>Take care of multiple Indirection, Return address
                !>>>      of data to be fetched.

(!START ADR.FETCH

        (IF ((TMPADR<0> EQL 1) AND (EM<0> EQL 0)) =>    !Indirect?
            (
LOOP1:=         (    (READ.Memory)NEXT
                     (TMPINDIRECT←TMPMDR<0>)NEXT !Save indirect bit
!see p 2-3 "How to Use the Nova Computers" Rev. 09, 1974 *wd15 7/1/77
                     (IF (TMPADR<1:11> EQL 1) =>
        !Is this an Increment or Decrement Memory location?
                         (INCR.DECR.Memory)           !YES
                     )NEXT !END IF (TMPADR<0:11> EQL 1)
                     (TMPADR<0:15>←TMPMDR<0:15>)NEXT
                     (IF (TMPINDIRECT EQL 1) =>  !More Indirection?
                         (
                             (LOOP1)
                         )
                     )!END IF (TMPMDR<0> EQL 1)
                )!END LOOP1
            )
        )!END IF TMPADR<0> EQL 1

);!END ADR.FETCH


!A D R . S E T U P      Uses Variables  MDR<5:15>, TMPADR<0:15>
                                 !, AC2<1:15>, AC3<1:15>, PC<0:15>.
!A D R . S E T U P      Uses NO Routines
!A D R . S E T U P


ADR.SETUP:=             !>>>For multiple Indirection.

(!START ADR.SETUP
           (DECODE MDR<6:7> =>              !Decode Index Field
                (    (TMPADR<0:15>←MDR<0:15>)
                );       !00= Page zero Addressing
                (    (TMPADR<1:15>←MDR<0:15>)NEXT
                     (IF (MDR<0> EQL 1) =>
                  (THPADR(0:7)<-"FF)
               )NEXT !END IF (MDR<0> EQL 1)
               (THPADR<0:15>-(PC<0:15> + THPADR<0:15>)<15:0>)
            );!01= PC relative Addressing  (with sign extention)
            (  (THPADR<1:15>-MDR<0:15>)NEXT
               (IF (MDR<0> EQL 1) =>
                  (THPADR<0:7>-"FF)
               )NEXT !END IF (MDR<0> EQL 1)
               (THPADR<0:15>-(AC2<0:15> + THPADR<0:15>)<15:0>)
            );!10= Index with AC2 Addressing. (with sign extention)
            (  (THPADR<1:15>-MDR<0:15>)NEXT
               (IF (MDR<0> EQL 1) =>
                  (THPADR<0:7>-"FF)
               )NEXT !END IF (MDR<0> EQL 1)
               (THPADR<0:15>-(AC3<0:15> + THPADR<0:15>)<15:0>)
            )!11= Index with AC3 Addressing. (with sign extention)
         )NEXT !END DECODE MDR<6:7>
         (IF MDR<5> =>                     !Inderection?
            (  (READ.Memory)NEXT
               (IF (THPADR<1:11> EQL 1) => !Is this an Increment
                           !or Decrement Memory location?
                  (INCR.DECR.Memory)              !YES
               )NEXT !END IF (THPADR<0:11> EQL 1)
               (THPADR<0:15>-THPMDR<0:15>)NEXT
               (ADR.FETCH)
            )
         )!END IF MDR<5>
);!END ADR.SETUP


!S K I P  Uses Variables  MDR<13:15>, TMP.NO.OP<0>
                        ! PC<0:15>, TMP.SHIFTER<0:16>.
!S K I P  Uses NO Routines.
!S K I P


SKIP:=    !>>> Handles the Skip operation specified by the SKIP field.

(!START SKIP

   (DECODE MDR<13:15> =>                    !Skip field
      (  (PC<0:15>-(PC<0:15> + 1)<15:0>)
      );                    !000= Never Skip
      (  (PC<0:15>-(PC<0:15> + 2)<15:0>)
      );                    !001= Skip always
      (  (DECODE (TMP.SHIFTER<0> EQL 0) =>
            (PC<0:15>-(PC<0:15> + 1)<15:0>);
            (PC<0:15>-(PC<0:15> + 2)<15:0>)
         )!END DECODE
      );                    !010= Skip if Carry = 0.
      (  (DECODE (TMP.SHIFTER<0> NEQ 0) =>
            (PC<0:15>-(PC<0:15> + 1)<15:0>);
            (PC<0:15>-(PC<0:15> + 2)<15:0>)
         )!END DECODE
      );                    !011= Skip if Carry = NOT 0.
      (  (DECODE (TMP.SHIFTER<1:16> EQL 0) =>
            (PC<0:15>-(PC<0:15> + 1)<15:0>);
            (PC<0:15>-(PC<0:15> + 2)<15:0>)
         )!END DECODE
      );                    !100= Skip if Result= 0.
      (  (DECODE (TMP.SHIFTER<1:16> NEQ 0) =>
            (PC<0:15>-(PC<0:15> + 1)<15:0>);
            (PC<0:15>-(PC<0:15> + 2)<15:0>)
         )!END DECODE
      );                    !101= Skip if Result = NOT 0.
      (  (DECODE ((TMP.SHIFTER<0> EQL 0) OR
                  (TMP.SHIFTER<1:16> EQL 0))=>
            (PC<0:15>-(PC<0:15> + 1)<15:0>);
            (PC<0:15>-(PC<0:15> + 2)<15:0>)
         )!END DECODE
      );                    !110= Skip if Carry OR Result= 0.
      (  (DECODE ((TMP.SHIFTER<0> NEQ 0) AND
                  (TMP.SHIFTER<1:16> NEQ 0))=>
            (PC<0:15>-(PC<0:15> + 1)<15:0>);
            (PC<0:15>-(PC<0:15> + 2)<15:0>)
         )!END DECODE
      )                     !111= Skip if Carry AND Result= NOT 0.
   )!END DECODE MDR<13:15>

);!END SKIP


!N O L O A D . L O A D  Uses Variables  MDR<12>, CARRY<0>,
                                  ! TMP.SHIFTER<0:16>, AC[0:3]<0:15>.
!N O L O A D . L O A D  Uses NO Routines.
!N O L O A D . L O A D
```

```
NOLOAD.LOAD:=               !>>> Load the Destination and Carry if true.

(!START NOLOAD.LOAD

    (IF MDR<12> EQL 0 =>               !Do we load?
        (    (CARRY<0>←TMP.SHIFTER<0>);                  !YES
             (AC[MDR<3:4>]<0:15>←TMP.SHIFTER<1:16>)
        )
    )!END IF MDR<12> EQL 0

);!END NOLOAD.LOAD


!S H I F T             Uses Variables  MDR<8:9>, TMP.SHIFTER<0:16>,
!                                                tmp.fctn.opt<0:16>
!S H I F T             Uses NO Routines
!S H I F T


SHIFT:=          !>>> Take care of shift determined by shift op-code (SH)

(!START SHIFT

    (DECODE MDR<8:9> =>                       !SH field
        (    (TMP.SHIFTER<0:16>←tmp.fctn.opt<0:16>)
        );                                   !00=NO SHIFT
        (    (TMP SHIFTER<0:16>←tmp.fctn.opt<0:16> †RL 1)
        );                                   !01=SHIFT left 1 possition
        (    (TMP.SHIFTER<0:16>←tmp.fctn.opt<0:16> †RR 1)
        );                                   !01=SHIFT right 1 possition
        (    (TMP.SHIFTER<0>←tmp.fctn.opt<0>);
             (TMP.SHIFTER<1:8>←tmp.fctn.opt<9:16>);
             (TMP.SHIFTER<9:16>←tmp.fctn.opt<1:8>)
        )                                    !11=SWAP bytes and pass Carry
    )!END DECODE MDR<8:9>

);!END SHIFT


!A N D D        Uses Variables  tmp.fctn.opt<0:16>, CARRY<0>,
!                                           AC[0:3]<0:15>, MDR<1:4>,
!A N D D        Uses NO Routines.
!A N D D


ANDD:=                      !>>> AND Source and Destination.
                            !>>>    Pass answer and Carry bit to Shifter

(!START ANDD

    tmp.fctn.opt<0:16>←CARRY<0>@(AC[MDR<1:2>]<0:15> AND
                                 AC[MDR<3:4>]<0:15>)

);!END ANDD


!A D D  Uses Variables  tmp.fctn.opt<0:16>, AC[0:3]<0:15>, MDR<1:4>,
!                       CARRY<0>, OVF<0>, TMP0REG<0:15>, TMP1REG<0:15>,
!A D D  Uses NO Routines.
!A D D


ADD:=                       !>>> ADD the Source to the Destination and
                            !>>>    take care of Carry and OVF bits. Make
                            !>>>    the result available to Shifter.

(!START ADD

    (TMP0REG<0:15>←AC[MDR<1:2>]<0:15>);
    (TMP1REG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
    (tmp.fctn.opt<0:16>←TMP0REG<0:15> + TMP1REG<0:15>)NEXT
    (IF (((TMP0REG<0> EQL TMP1REG<0>) AND
         (TMP0REG<0> NEQ tmp.fctn.opt<1>))=>
        (OVF<0>←1)                              !Take care of OVerFlow
    );!END IF
    (DECODE tmp.fctn.opt<0> =>              !Take care of the Carry bit.
        (tmp.fctn.opt<0>←CARRY<0>);
        (tmp.fctn.opt<0>← NOT (CARRY<0>))
    )!END DECODE tmp.fctn.opt<0>

);!END ADD


!S U B T R A C T       Uses Variables  tmp.fctn.opt<0:16>, AC[0:3]<0:15>,
!                  MDR<1:4>, CARRY<0>, OVF<0>, TMPOREG<0:15>, TMPIREG<0:15>.
!S U B T R A C T          Uses NO Routines.
!S U B T R A C T


SUBTRACT:=                    !>>> Subtract the source from the Destination
                              !>>>     and take care of the Carry bit. Make
                              !>>>     the result available to the Shifter.
(!START SUBTRACT

    (TMPOREG<0:15>←AC[MDR<1:2>]<0:15>);
    (TMPIREG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
    (tmp.fctn.opt<0:16>←(TMPIREG<0:15> +
                          (NOT(TMPOREG<0:15>) + 1))<16:0>)NEXT
    (IF ((TMPOREG<0> NEQ TMPIREG<0>) AND
               (TMPOREG<0> EQL tmp.fctn.opt<1>))=>
        (OVF<0>←1)                         !Take care of OVerFlow
    );!END IF
    (DECODE tmp.fctn.opt<0> =>                !Take care of Carry bit.
        (tmp.fctn.opt<0>←CARRY<0>);
        (tmp.fctn.opt<0>← NOT (CARRY<0>))
    )!END DECODE tmp.fctn.opt<0>

);!END SUBTRACT


!A D D C O M P L E M E N T      Uses Variables  tmp.fctn.opt<0:16>,
!                               AC[0:3]<0:15>, MDR<1:4>, CARRY<0>,
!                               OVF<0>, TMPOREG<0:15>, TMPIREG<0:15>.
!A D D C O M P L E M E N T      Uses NO Routines.
!A D D C O M P L E M E N T


ADDCOMPLEMENT:=               !>>> Complement the Source then
                              !>>> Add it to the Destination and
                              !>>> make it available to the Shifter.
                              !>>> Also take care of the Carry and OVerFlow.

(!START ADDCOMPLEMENT

    (TMPOREG<0:15>←AC[MDR<1:2>]<0:15>);
    (TMPIREG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
    tmp.fctn.opt<0:16>←( NOT (TMPOREG<0:15>)) + TMPIREG<0:15> NEXT
    (IF ((TMPOREG<0> NEQ TMPIREG<0>) AND
               (TMPOREG<0> EQL tmp.fctn.opt<1>))=>
        (OVF<0>←1)                         !Take care of OVerFlow
    );!END IF
    (DECODE tmp.fctn.opt<0> =>                !Take care of the Carry bit.
        (tmp.fctn.opt<0>←CARRY<0>);
        (tmp.fctn.opt<0>← NOT(CARRY<0>))
    )!END DECODE tmp.fctn.opt<0>

);!END ADDCOMPLEMENT


!I N C R E M E N T      Uses Variables  tmp.fctn.opt<0:16>, AC[0:3]<0:15>,
!                       MDR<1:2>, CARRY<0>, OVF<0>, TMPOREG<0:15>.
!I N C R E M E N T      Uses NO Routines.
!I N C R E M E N T


INCREMENT:=                   !>>> Increment the Source, and pass it and
                              !>>>     the resultant Carry to the Shifter.

(!START INCREMENT

    (TMPOREG<0:15>←AC[MDR<1:2>]<0:15>)NEXT
    tmp.fctn.opt<0:16>←TMPOREG<0:15>+1 NEXT
    (IF (TMPOREG<0:15> EQL #77777)=>     !Will an OVerFlow occure?
        (OVF<0>←1)                         !yes
    );!END IF (TMPOREG<0:15> EQL #77777)
    (DECODE tmp.fctn.opt<0> =>                !Take care of the Carry bit.
        (tmp.fctn.opt<0>←CARRY<0>);
        (tmp.fctn.opt<0>← NOT (CARRY<0>))
    )!END DECODE tmp.fctn.opt<0>

);!END INCREMENT


!       M O V E        Uses Variables  tmp.fctn.opt<0:16>, CARRY<0>,
!                                      AC[1:2]<0:15>, MDR<1:2>.
!       M O V E        Uses NO Routines.
!       M O V E


MOVE:=                        !>>> Move the Source and Carry to the Shifter.
```

```
(!START MOVE

    tmp.fctn.opt<0:16>←CARRY<0>@AC[MDR<1:2>]<0:15>

)!!END MOVE


!N E G A T E    Uses Variables  tmp.fctn.opt<0:16>, AC[1:2]<0:15>,
!                         MDR<1:2>, CARRY<0>, OVF<0>, TMPOREG<0:15>.
!N E G A T E    Uses NO Routines.
!N E G A T E


NEGATE:=                  !>>> Negate the source and put in tmp.fctn.opt

(!START NEGATE

    (TMPOREG<0:15>←AC[MDR<1:2>]<0:15>)NEXT
    (tmp.fctn.opt<0:16>←( NOT (TMPOREG<0:15>)) + 1)NEXT
    (IF (TMPOREG<0:15> EQL '1000000000000000)=>     !Will we get an overflow?
        (OVF<0>←1)                          !yes
    );!END IF
    (DECODE tmp.fctn.opt<0> =>              !Take care of the Carry bit.
        (tmp.fctn.opt<0>←CARRY<0>);
        (tmp.fctn.opt<0>← NOT (CARRY<0>))
    )!END DECODE tmp.fctn.opt<0>

);!END NEGATE


!C O M P L E M E N T    Uses Variables  tmp.fctn.opt<0:16>, AC[1:2]<0:15>,
!                                              MDR<1:2>, CARRY<0>.
!C O M P L E M E N T    Uses NO Routines.
!C O M P L E M E N T


COMPLEMENT:=              !>>> Complement the Source and put the
                          !>>>  result and the Carry bit at the
                          !>>>  input of the Shifter.

(!START COMPLEMENT

    tmp.fctn.opt<0:16>←CARRY<0>@( NOT (AC[MDR<1:2>]<0:15>))

)!!END COMPLEMENT


!C A R R Y . S E T U P  Uses Variables  MDR<10:11>, TMP.NO.OP<0>, CARRY<0>.
!C A R R Y . S E T U P           Uses NO Routines.
!C A R R Y . S E T U P


CARRY.SETUP:=             !>>> Initialize the Carry bit.

(!START CARRY.SETUP

   (DECODE MDR<10:11> =>             !Decode set up options.
        (NOP);                          !00=Leave as is.
        (CARRY<0>←0);                   !01=Clear initially.
        (CARRY<0>←1);                   !10=Set initially.
        (CARRY<0>←( NOT (CARRY<0>)))    !11=Complement its present value.
    )!END DECODE MDR<10:11>

)!!END CARRY.SETUP
!       SOME OF THOSE RESOURCE MANAGEMENT INSTRUCTIONS

PUSH.STACK:=BEGIN        !PUSH TMPMDR ONTO STACK
        TMPADR ← (SP - 1)<15:0> NEXT
        SP ← TMPADR NEXT
        (DECODE TMPADR<0:15> LSS #420 =>
        \0      WRITE.MEMORY;
        \1      BEGIN
                ION<0>←0; TMPMDR←PC; TMPADR←#44 NEXT
                WRITE.MEMORY NEXT
                TMPADR←#45 NEXT
                READ.MEMORY NEXT
                PC←TMPMDR NEXT
                BAILOUT IEXEC
                END
        )
        END;

SYSTRAP:=BEGIN  !SYSTEM TRAP SEQUENCE
        TEMP1 ← MSR ;
        TEMP2 ← SP ;
        TEMP3 ← SL NEXT
        (IF NOT XMD =>
                XMD ← 1 ; SXMD ← 1 ; XMDC ← 0 NEXT
                TMPADR ← 3 NEXT
                READ.MEMORY NEXT
                TMPADR ← (TMPMDR + 4)<15:0> NEXT
                READ.MEMORY NEXT
                SP ← TMPMDR NEXT
                TMPADR ← (TMPADR + 1)<15:0> NEXT
                READ.MEMORY NEXT
                SL ← TMPMDR
        ) NEXT
        TMPMDR ← TEMP3 NEXT
        PUSH.STACK NEXT
        TMPMDR ← TEMP2 NEXT
        PUSH.STACK NEXT
        TMPMDR ← TEMP1 NEXT
        PUSH.STACK NEXT
        TMPMDR ← PC NEXT
        PUSH.STACK NEXT
        TMPADR ← 5 NEXT
        READ.MEMORY NEXT
        PUSH.STACK NEXT
        TMPADR ← 2 NEXT
        READ.MEMORY NEXT
        TMPADR ← (TMPMDR + TRAP.INDEX<3:0>)<15:0> NEXT
        READ.MEMORY NEXT
        TMPADR ← TMPMDR NEXT
        ADR.FETCH NEXT
        PC ← TMPMDR
        END;

CKPRV:= BEGIN
        IF XMDC =>
                PRPE ← 1 NEXT
                TRAP.INDEX ← 2 NEXT
                SYSTRAP NEXT
                BAILOUT ARITHMETIC.OR.LOGIC
        END;

WRMAP:= BEGIN
        CKPRV NEXT
        TEMP1 ← AC0; TEMP2 ← AC1; TMPADR ← AC2 NEXT
        (DECODE TEMP1<6> =>
                TEMP1 ← TEMP1<2:0>*64 + TEMP1<15:10>;
                (BAILOUT WRMAP)
        ) NEXT
WRMAP1:=(IF TEMP2 GTR 0 =>
                READ.MEMORY NEXT
                MAPREG[TEMP1<8:0>] ← TMPMDR NEXT
                TMPADR ← (TMPADR + 1)<15:0> ;
                TEMP1 ← (TEMP1 + 1)<15:0> ;
                TEMP2 ← (TEMP2 - 1)<15:0> NEXT
                WRMAP1
        )
        END;

RDMAP:= BEGIN   !READ MAP FILE
        CKPRV NEXT
        TEMP1 ← AC0; TEMP2 ← AC1; TMPADR ← AC2 NEXT
        (DECODE TEMP1<6> =>
                TEMP1 ← TEMP1<2:0>*64 + TEMP1<15:10>;
                (BAILOUT RDMAP)
        ) NEXT
RDMAP1:=(IF TEMP2 GTR 0 =>
                TMPMDR ← MAPREG[TEMP1<8:0>] NEXT
                WRITE.MEMORY NEXT
                TMPADR ← (TMPADR + 1)<15:0> ;
                TEMP1 ← (TEMP1 + 1)<15:0> ;
                TEMP2 ← (TEMP2 - 1)<15:0> NEXT
                RDMAP1
        )
        END;

WRWRD:= BEGIN   !WRITE SINGLE WORD
        CKPRV NEXT
        TMPAREG←AC0 NEXT
        MAPREG[(TMPAREG<13:15>*64+TMPAREG<0:5>)<8:0>]<0:15> ← AC1<0:15>
        END;

RDWRD:= BEGIN   !READ SINGLE WORD
        CKPRV NEXT
        TMPAREG←AC0 NEXT
        AC1<0:15> ← MAPREG[(TMPAREG<13:15>*64+TMPAREG<0:5>)<8:0>]<0:15>
        END;
```

```
WMSR:=  BEGIN   !WRITE MAP STATUS REGISTER
        CKPRV NEXT
        UEM ← AC[MDR<3:4>]<2>;
        MSR<4:15> ← AC[MDR<3:4>]<4:15> NEXT
        MVR<13:15> ← MSR<13:15>
        END;

RMSR:=  BEGIN   !READ MAP STATUS REGISTER
        CKPRV NEXT
        AC[MDR<3:4>]<0:15> ← MSR<0:15>
        END;

RMVR:=  BEGIN   !READ MAP VIOLATION REGISTER
        CKPRV NEXT
        AC[MDR<3:4>]<0:15> ← MVR<0:15>
        END;

CMVR:=  BEGIN   !CLEAR MAP VIOLATION REGISTER
        CKPRV NEXT
        MVR<1:7> ← 0
        END;

CDMA:=  BEGIN   !CLEAR DMA VIOLATION
        CKPRV NEXT
        MVR<0> ← 0
        END;

RLAF:=  BEGIN   !READ LAST ADDRESS FILE
        CKPRV NEXT
        NOP
        END;

EXMAP:= BEGIN   !ENABLE EXECUTIVE DATA MAP
        CKPRV NEXT
        XMD ← 1 ; SXMD ← 1 ; XMDC ← 0
        END;

DXMAP:= BEGIN   !DISABLE EXECUTIVE DATA MAP
        CKPRV NEXT
        XMD ← 0 ; SXMD ← 0 ; XMDC ← 1
        END;

MAPSI:= BEGIN   !MAP SINGLE INSTRUCTION
        CKPRV NEXT
        MSI ← 1 NEXT
        BAILOUT IEXEC
        END;

MAPSD:= BEGIN   !MAP SINGLE DATA
        CKPRV NEXT
        MSD ← 1 NEXT
        BAILOUT IEXEC
        END;

RMST:=  BEGIN   !READ REMOTE MEMORY CHASSIS STATUS
        CKPRV NEXT
        NOP
        END;

UJMP:=  BEGIN   !EXECUTIVE TO USER JUMP
        CKPRV NEXT
        XMDC ← 0 ; XMD ← 1 ; SXMD ← 1 ;
        EM ← UEM ;
        MVR<1:7> ← 0 ;
        PC<0:15> ← AC[MDR<3:4>]<0:15>
        END;

EUB:=   BEGIN   !EXECUTIVE TO USER BRANCH
        CKPRV NEXT
        TMPADR<0:15> ← AC0<0:15> NEXT
        READ.MEMORY NEXT
        PC<0:15> ← TMPMDR<0:15> NEXT
        TMPADR<0:15> ← (TMPADR<0:15> + 1)<15:0> NEXT
        READ.MEMORY NEXT
        SP<0:15> ← TMPMDR<0:15> NEXT
        TMPADR<0:15> ← (TMPADR<0:15> + 1)<15:0> NEXT
        READ.MEMORY NEXT
        MSR<0:15> ← TMPMDR<0:15> NEXT
        TMPADR<0:15> ← (TMPADR<0:15> + 1)<15:0> NEXT
        READ.MEMORY NEXT
        SL<0:15> ← TMPMDR<0:15> NEXT
        XMDC ← 1 ; XMD ← 0 ; SXMD ← 0 NEXT
        MVR<1:7> ← 0 NEXT
        EM ← UEM
        END;
```

```
ECALL:= BEGIN    !EXECUTIVE CALL
        TRAP.INDEX ← 1 NEXT
        SYSTRAP
        END;

TRAP:=  BEGIN    !SYSTEM TRAP
        TRAP.INDEX ← AC(MDR<3:4>)<12:15> NEXT
        SYSTRAP
        END;
!       A R I T H M E T I C . o r . L O G I C   Uses Variables  MDR<5:7>.
!       A R I T H M E T I C . o r . L O G I C   Uses Routines   CARRY.SETUP, COMPLEMENT, NEGATE, MOVE,
!                                                        INCREMENT, ADDCOMPLEMENT, SUBTRACT, ADD, ANDD,
!                                                        SHIFT, NOLOAD.LOAD, SKIP.
!       A R I T H M E T I C . o r . L O G I C


ARITHMETIC.or.LOGIC:=     !>>> Take care of Arithmetic or Logic functions and Increment PC.

(!START ARITHMETIC.or.LOGIC

    (if(mdr<1:15> eql #03240)=> !WDI5 7/12/77
FNG:= (!start floating point negate (=ADDDR 0,0)
        tmpdoublereg<0:32>←(not(ac0<0:15>□ac1<0:15>)+1)<31:0> next
        ac0<0:15>←tmpdoublereg<1:16>;
        ac1<0:15>←tmpdoublereg<17:32> next
        INCR.PC NEXT
        bailout arithmetic.or.logic
      )!end floating point negate
    ) next !end if


!===========RMU Resource Management Unit WDI5 7/13/77============


    (if(mdr<1> eql 0) and (mdr<5:15> eql #1130)=>
      begin !RMU decode WDI5  7/13/77
        INCR.PC NEXT
        (decode mdr<2:4>=>
          WRMAP;
          RDMAP;
          WRWRD;
          RDWRD;
          CHVR;
          CDMA;
          MAPSI;
          MAPSD
        ) next !end decode
        bailout arithmetic.or.logic
      end !RMU decode
    ) next !end if


    (if mdr<5:15> eql #1110=>
      begin !RMU decode WDI5 7/13/77
        INCR.PC NEXT
        (decode mdr<1:2>=>
          RMSR;
          WMSR;
          RMVR;
          UJMP
        ) next !end decode
        bailout arithmetic.or.logic
      end !RMU decode
    ) next !end if


    (if(mdr<1:2> eql '10) and (mdr<5:15> eql #11300)=>
      begin !RMU decode WDI5 7/13/77
        INCR.PC NEXT
        (decode mdr<3:4>=>
          IXMAP;
          DXMAP;
          RMST;
          nop              !unused
        ) next !end decode
        bailout arithmetic.or.logic
      end !RMU decode
    ) next !end if


    (if(mdr<1:2> eql '01) and (mdr<5:15> eql #0110)=>
      begin !RMU decode WDI5 7/13/77
        TRAP next
        bailout arithmetic.or.logic
      end !RMU decode
    ) next !end if
```

```
!-----------end of PMU decode********************************

    (CARRY.SETUP)NEXT                         !Take care of setting up the Carry bit.
    (DECODE MDR<5:7> =>                       !Decode op-code determining Function Generator
                                              !    action (function stores answer in tmp.fctn.opt
                                              !    and takes care of Carry bit.
        (COMPLEMENT);                             !000=Complement Source
        (NEGATE);                                 !001=Negate Source
        (MOVE);                                   !010=Move Source
        (INCREMENT);                              !011=Increment Source
        (ADDCOMPLEMENT);                          !100=Add the Complemented Source to the Destination
        (SUBTRACT);                               !101=Subtact the Source from the Destination
        (ADD);                                    !110=Add the Source to the Destination
        (ANDD)                                    !111=AND the Source to the Destination
    )NEXT !END DECODE MDR<5:7>
    (SHIFT)NEXT                               !Take care of Shifter op-code
    (NOLOAD.LOAD)NEXT                         !Load the Destination if we are suppose to.
    (SKIP)                                    !Take care of Skip op-code and Increment Program Counter

));!END ARITHMETIC.or.LOGIC
!
!I/O INSTRUCTIONS
!
INDEV:= (AC[MDR<3:4>] ← DEV.INREG);

OUTDEV:=(DEV.OUTREG ← AC[MDR<3:4>]);


!     I N P U T . O U T P U T
!     I N P U T . O U T P U T
!     I N P U T . O U T P U T


INPUT.OUTPUT:=              !>>> I/O stuff. (and all the hacks)

!!!!!!!! This is the section of the machine that was thrown in after
!!!!!!!!the original NOVA was designed.  As such, it is a very
!!!!!!!!dirty part of the ISP discription.


((START INPUT.OUTPUT

!START MEMORY TO ACCUMULATOR INSTRUCTIONS
        (IF((MDR<5:6> NEQ '11) AND ((MDR<7>@MDR<10:15>) EQL '1000001))=>
          (
            (IF MDR<5:6> EQL '00 =>          !NEXT ADDRESS
              ( (TMPADR<0:15>←(TMPADR<0:15> + 1)<15:0>)NEXT
                (READ.Memory)NEXT
                (TMPADR<0:15>←TMPMDR<0:15>)NEXT
                (R.DATA)
              )
            )NEXT !END IF MDR<5:6> EQL '00

            (IF MDR<5:6> EQL '01 =>          !NEXT ADDRESS INDEXED
              ( (TMPADR<0:15>←(TMPADR<0:15> + 1)<15:0>)NEXT
                (READ.Memory)NEXT
                (TMPADR<0:15>←(TMPMDR<0:15> + AC2<0:15>)<15:0>)NEXT
                (R.DATA)
              )
            )NEXT !END IF MDR<5:6> EQL '01

            (IF MDR<5:6> EQL '10 =>          !NEXT WORD
              ( (TMPADR<0:15>←(TMPADR<0:15> + 1)<15:0>)NEXT
                (READ.Memory)
              )
            )NEXT !END IF MDR<5:6> EQL '10



            (DECODE MDR<8:9> =>

LDFN:=        (                                                                   !LDFN   LOAD FROM NEXT
                (AC[MDR<3:4>]<0:15>←TMPMDR<0:15>)
              );
ADFN:=        (                                                                   !ADFN   ADD f-om next


                (TMPIREG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
                (IF ((TMPIREG<0> EQL TMPMDR<0>) AND (TMPMDR<0> NEQ (TMPMDR<0:15> + TMPIREG<0:15>)<15>))=>
                  (OVF<0>←1)                    !Yes we have an OVerflow
                );
                (AC[MDR<3:4>]<0:15>←(TMPMDR<0:15> + TMPIREG<0:15>)<15:0>)
              ));
SBFN:=         (                                                              !SBFN   SUBTRACT from next word
                 (TMPIREG<0:15>+AC[MDR<3:4>]<0:15>)NEXT
                 (IF ((TMPMDR<0> NEQ TMPIREG<0>) AND (TMPMDR<0> EQL (TMPIREG<0:15>   TMPMDR<0:15>)<15:0>))=> !OVerFlow?
                   (OVF<0>+1)                                !Take care of OVerFlow
                 );
                 (AC[MDR<3:4>]<0:15>+(TMPIREG<0:15>   TMPMDR<0:15>)<15:0>)
               );
ANFN:=         (                                                              !ANFN   AND from next
                 (AC[MDR<3:4>]<0:15>+AC[MDR<3:4>]<0:15> AND TMPMDR<0:15>)
               )
             );!END DECODE MDR<8:9>
             (PC<0:15>+(PC<0:15> + 2)<15:0>) NEXT
             BAILOUT INPUT,OUTPUT
           )
         ) NEXT !end if
!END MEMORY TO ACCUMULATOR INSTRUCTIONS
!START ACCUMULATOR TO MEMORY INSTRUCTIONS
         (IF((MDR<5>@MDR<7>@MDR<10:15>)) EQL '01000000)=>
           (
           (DECODE MDR<6> =>          !INDEXED?
             ( (TMPADR<0:15>+(TMPADR<0:15> + 1)<15:0>)NEXT
               (READ.Memory)NEXT
               (TMPADR<0:15>+TMPMDR<0:15>)
             );         .       !NOT INDEXED

             ( (TMPADR<0:15>+(TMPADR<0:15> + 1)<15:0>)NEXT
               (READ.Memory)NEXT
               (TMPADR<0:15>+(TMPMDR<0:15> + AC2<0:15>)<15:0>)
             )             !INDEXED ON AC2
           )NEXT !END DECODE MDR<6>


           (DECODE MDR<8:9> =>)
STTN:=       (                                                                !STTN   store AC to next
               (TMPMDR<0:15>+AC[MDR<3:4>]<0:15>)NEXT
               (W.DATA)
             );
ADTN:=       (                                                                !ADTN   store AC+contents of next
               (R.DATA)NEXT                                           !         address to next
               (TMPIREG<0:15>+AC[MDR<3:4>]<0:15>)NEXT
               (IF ((TMPIREG<0> EQL TMPMDR<0>) AND (TMPMDR<0> NEQ (TMPMDR<0:15> + TMPIREG<0:15>)<15>))=>
                 (OVF<0>+1)           !Yes we have an OVerFlow
               );
               (TMPMDR<0:15>+(TMPIREG<0:15>+TMPMDR<0:15>)<15:0>)NEXT
               (W.DATA)
             );
MGTN:=       (                                                                !MGTN   Merge to next
               (R.DATA)NEXT
               (TMPIREG<0:15>+AC1<0:15>)NEXT
               (TMPMDR<0:15>+((TMPIREG<0:15> AND AC[MDR<3:4>]<0:15>) OR (TMPMDR<0:15> AND (NOT(TMPIREG<0:15>)))))NEXT
               (W.DATA)
             );
ANTNA:=      (                                                                !ANTNA  AND to next address
               (R.DATA)NEXT
               (TMPMDR<0:15>+(AC[MDR<3:4>]<0:15> AND TMPMDR<0:15>))NEXT
               (W.DATA)
             )
           )NEXT !END DECODE MDR<8:9>
           (PC<0:15>+(PC<0:15> + 2)<15:0>)
           NEXT BAILOUT INPUT,OUTPUT
           )
         ) NEXT !end if
!END ACCUMULATOR TO MEMORY INSTRUCTION
!START DOUBLE-PRECISION INSTRUCTIONS
         (IF((MDR<5:7>@MDR<10:15>)) EQL '010011111)=>
           (
             (DECODE MDR<9> =>        !INDEXED?
               ( (TMPADR<0:15>+(TMPADR<0:15> + 1)<15:0>)NEXT
                 (READ.Memory)NEXT
                 (TMPADR<0:15>+TMPMDR<0:15>)
               );                 !NOT INDEXED
               ( (TMPADR<0:15>+(TMPADR<0:15> + 1)<15:0>)NEXT
                 (READ.Memory)NEXT
                 (TMPADR<0:15>+(TMPMDR<0:15> + AC2<0:15>)<15:0>)
               )             !INDEXED ON AC2
             )NEXT !END DECODE MDR<9>


             (DECODE MDR<3:4> =>)
DST:=          ( !DST    Double Store (FST also)
                 (TMPMDR<0:15>+AC0<0:15>)NEXT
                 (W.DATA)NEXT
                 (TMPADR<0:15>+(TMPADR<0:15>+1)<15:0>);
                 (TMPMDR<0:15>+AC1<0:15>)NEXT
                 (W.DATA)
               );
```

```
DADD:=          ( !DADD    [illegible] ADD
                  [illegible]                 !NOTS 229-22
                  [illegible] NEXT
                  [illegible]
                  [illegible] NEXT
                  [illegible]
                  (TMPDOUBLEREG<0:32>+(AC1<0:15> + TMPMDR<0:15> + TMPDOUBLEREG<0:32>)<32:0>)NEXT
                  (AC0<0:15>+TMPDOUBLEREG<1:16>);
                  (AC1<0:15>+TMPDOUBLEREG<17:32>)
                );
DNA:=          ; ( !DNA     Double NEGATE and ADD
                  (P.DATA)NEXT
                  (TMPDOUBLEREG<0:32>+((NOT(AC0<0:15>@AC1<0:15>))<31:0>)) + 1<31:0>)NEXT
                  (TMPDOUBLEREG<0:16>+(TMPDOUBLEREG<0:16> + TMPMDR<0:15>)<16:0>)NEXT
                  (TMPMDR<0:15>+(TMPADR<0:15> + 1)<15:0>)NEXT
                  (P.DATA)NEXT
                  (TMPDOUBLEREG<0:32>+(TMPMDR<0:15> + TMPDOUBLEREG<0:32>)<32:0>)NEXT
                  (AC0<0:15>+TMPDOUBLEREG<1:16>);
                  (AC1<0:15>+TMPDOUBLEREG<17:32>)
                ));
DLD:=           ( !DLD     Double Load (FLD also)
                . (P.DATA)NEXT
                  (AC0<0:15>+TMPMDR<0:15>)NEXT
                  (TMPADR<0:15>+(TMPADR<0:15>+1)<15:0>)NEXT
                  (P.DATA)NEXT
                  (AC1<0:15>+TMPMDR<0:15>)
                )
              );!END DECODE MDR<3:4>
              (PC<0:15>+(PC<0:15> + 2)<15:0>)
              NEXT BAILOUT INPUT,OUTPUT
            )
          ) NEXT !end if
!END DOUBLE PRECISION INSTRUCTIONS
!START ARITHMETIC SHIFT; SHIFT; ADD IMMEDIATE INSTRUCTIONS
         (IF(MDR<3:7> EQL '01000)=>
           (
              (DECODE MDR<8:9> =>
ADPI:= \00      ( (AC2<0:15>+(AC2<0:15> + MDR<10:15>)<15:0>)                        !ADPI   ADD POSITIVE IMMEDIATE
                );
\01             (DECODE MDR<10:11> =>
LDSHD:= \01 \00 ( (TMP1REG<0:15>+AC0<0:15>)NEXT                                    !LDSHD  Left dual-mode shift double
                   (AC1<0:15>+TMP1REG<0:15> !SL0 MDR<12:15>);
                   (AC0<0:15>+TMP1REG<0:15> !RL MDR<12:15>)
                 );
LLSHD:= \01 \01 ( (TMPDOUBLEREG<1:32>+AC0<0:15>@AC1<0:15>)NEXT                     !LLSHD  Left logical shift double
                   (TMPDOUBLEREG<1:32>+TMPDOUBLEREG<1:32> !SL0 MDR<12:15>)NEXT
                   (AC0<0:15>+TMPDOUBLEREG<1:16>);
                   (AC1<0:15>+TMPDOUBLEREG<17:32>)
                 );
\01 \10          (DECODE MDR<12> =>
LASHD:= \01 \10 \0 ((TMPDOUBLEREG<1:32>+AC0<0:15>@AC1<0:15>)NEXT                   !LASHD  Left arithmetic shift double
                     (DECODE (((TMPDOUBLEREG<1:32> !SR0(31 - MDR<13:15>))EQL 0)OR(((TMPDOUBLEREG<1:32> !SR1(31 - MDR<13:15>))<31:0> + 1)<31:0> EQL
                       ( (CARRY<0>+1);
                         (OVF<0>+1)
                       );          !YES OVF and CARRY
                       ( (TMPDOUBLEREG<1:32>+TMPDOUBLEREG<1:32> !SL0 MDR<13:15>)NEXT
                         (AC0<0:15>+TMPDOUBLEREG<1:16>);
                         (AC1<0:15>+TMPDOUBLEREG<17:32>);
                         (CARRY<0>+0)
                       )           !NO OVF or CARRY
                     );!END DECODE (((TMPDOUBLEREG<1:32> !SR0(31 - MDR<13:15>)) etc.
                   );
RASHD:= \01 \10 \1 ((TMPDOUBLEREG<1:32>+AC0<0:15>@AC1<0:15>)NEXT                   !RASHD  Right arithmetic shift double
                     (DECODE TMPDOUBLEREG<1>=>          !Positive or negative
                       (TMPDOUBLEREG<1:32>+TMPDOUBLEREG<1:32> !SR0 MDR<13:15>);
                       (TMPDOUBLEREG<1:32>+TMPDOUBLEREG<1:32> !SR1 MDR<13:15>)
                     );!END DECODE TMPDOUBLEREG<1>
                     (AC0<0:15>+TMPDOUBLEREG<1:16>);
                     (AC1<0:15>+TMPDOUBLEREG<17:32>)
                   )
                 );!END DECODE MDR<12>
RLSHD:= \01 \11 ( (TMPDOUBLEREG<1:32>+AC0<0:15>@AC1<0:15>)NEXT                     !RLSHD  Right logical shift double
                   (TMPDOUBLEREG<1:32>+TMPDOUBLEREG<1:32> !SR0 MDR<12:15>)NEXT
                   (AC0<0:15>+TMPDOUBLEREG<1:16>);
                   (AC1<0:15>+TMPDOUBLEREG<17:32>)
                 )
               );!END DECODE MDR<10:11>
\10            (DECODE MDR<10:11>=>
LROT:= \10 \00  ( (AC0<0:15>+AC0<0:15> !RL MDR<12:15>)                             !LROT   Left rotate
                 );
LLSH:= \10 \01  ( (AC0<0:15>+AC0<0:15> !SL0 MDR<12:15>)                            !LLSH   Left logical shift
                 );
\10 \10          (DECODE MDR<12>=>
LASH:= \10 \10 \0 ( (TMPREG<0:15>+AC0<0:15>)NEXT                                   !LASH   Left arithmetic shift
                     (DECODE (((TMPREG<0:15> !SR0(15 - MDR<13:15>))EQL 0)OR(((TMPREG<0:15> !SR1(15 - MDR<13:15>))<15:0> + 1)<15:0> EQL 0))=>
                       ( (CARRY<0>+1);
                       [illegible]<0:15>)
                    [illegible]
                    ( [illegible] MDR<13:15> NEXT
                      [illegible]<0:15>)
                      [illegible]<0:0>)
                              [illegible]
                  ))!END DECODE [illegible]<0:15> [illegible]<0:15> MDR<13:15> [illegible]
                ));
RASH:= \10 \10 \1 ( (TMPOREG<0:15>←AC0<0:15>)NEXT                        !RASH   Right arithmetic shift
                  (DECODE TMPOREG<0>=>  !Positive or negative
                    ((TMPOREG<0:15>←TMPOREG<0:15> ISR0 MDR<13:15>);
                     (TMPOREG<0:15>←TMPOREG<0:15> ISR1 MDR<13:15>)
                  ));!END DECODE TMPOREG<0>
                  (AC0<0:15>←TMPOREG<0:15>)
                )
              ));!END DECODE MDR<12>
RLSH:= \10 \\1  ( (AC0<0:15>←AC0<0:15> ISR0 MDR<12:15>)                   !RLSH   Right logical shift
                )
             ));!END DECODE ADR<10:11>
ADNI:= \11      ( (AC2<0:15>←((AC2<0:15> + MDR<10:15>) MINUS '1000000)<15:0>)   !ADNI   Add negative immediate
                )
            ));!END DECODE MDR<8:9>
             (INCR.PC)
             NEXT BAILOUT INPUT.OUTPUT
           )
         ) NEXT !end if
!END ARITHMETIC SHIFT; SHIFT; ADD IMMEDIATE INSTRUCTIONS
!START PART 1 OF 4 SINGLE WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
         (IF((MDR<5:7>@MDR<10:15>) EQL '01000000)=>
           (
             (DECODE MDR<8:9>=>
RSP:=        ( (AC[MDR<3:4>]<0:15>←SP<0:15>)                       !RSP  READ STACK POINTER
               );
PSH:=          ( ((TMPADR<0:15>←(SP<0:15> - 1)<15:0>)NEXT          !PSH  PUSH ACCUMULATOR ONTO STACK
                 (SP<0:15>←TMPADR<0:15>));
                 (TMPMDR<0:15>←AC[MDR<3:4>]<0:15>)NEXT
                 (WRITE.Memory);
                 (IF TMPADR<0:15> LSS #420 =>       !Stack overflow?
                   ( (ION<0>←0);
                     (TMPMDR<0:15>←PC<0:15>);
                     (TMPADR<0:15>←#44)NEXT
                     (WRITE.Memory);
                     (TMPADR<0:15>←#45)NEXT
                     (READ.Memory)NEXT
                     (PC<0:15>←TMPMDR<0:15>)NEXT
                     (BAILOUT INPUT.OUTPUT)
                   )
                 )!END IF TMPADR<0:15> LSS #420
               ));
POP:=          ( (TMPADR<0:15>←SP<0:15>)NEXT                                !POP  Pop accumulator from stack


                 (READ.Memory);
                 (SP<0:15>←(TMPADR<0:15> + 1)<15:0>)NEXT
                 (AC[MDR<3:4>]<0:15>←TMPMDR<0:15>)
               );
WSP:=          ( (SP<0:15>←AC[MDR<3:4>]<0:15>)                       !WSP  Write stack pointer
               )
             ));!END DECODE MDR<8:9>
             (INCR.PC)
             NEXT BAILOUT INPUT.OUTPUT
           )
         ) NEXT !end if
!END PART 1 OF 4 SINGLE-WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
!START PART 2 OF 4 SINGLE-WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
         (IF((MDR<5:8>@MDR<10:15>) EQL '0101111111)=>
           (
             (DECODE MDR<9>=>
SMPY:=         ( (TMPOREG<0:15>←AC1<0:15>);                                !SMPY   Signed multiply
                 (TMPIREG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
                 (TMPOSIGN<0>←TMPOREG<0> XOR TMPIREG<0>)NEXT
                 (IF TMPOREG<0> EQL 1=>
                   (TMPOREG<0:15>←(MINUS TMPOREG<0:15>)<15:0>)
                 )!END IF TMPOREG<0> EQL 1
                 (IF TMPIREG<0> EQL 1=>
                   (TMPIREG<0:15>←(MINUS TMPIREG<0:15>)<15:0>)
                 )NEXT !END IF TMPIREG<0> EQL 1
                 (TMPDOUBLEREG<1:32>←TMPOREG<0:15> * TMPIREG<0:15>)NEXT
                 (IF TMPOSIGN<0> EQL 1=>
                   (TMPDOUBLEREG<1:32>←(MINUS TMPDOUBLEREG<1:32>)<31:0>)
                 )NEXT !END IF TMPOSIGN<0> EQL 1
                 (AC0<0:15>←TMPDOUBLEREG<1:16>);
                 (AC1<0:15>←TMPDOUBLEREG<17:32>)
               );
SDVD:=         ( (TMPDOUBLEREG<1:32>←AC0<0:15>@AC1<0:15>);                 !SDVD   Signed divide
                 (TMPIREG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
                (TMPOSIGN<0>←TMPDOUBLEREG<1> XOR TMPIREG<0>);
                (TMPISIGN<0>←TMPDOUBLEREG<1>)NEXT
                (IF TMPDOUBLEREG<1> EQL 1=>
                  (TMPDOUBLEREG<1:32>←(MINUS TMPDOUBLEREG<1:32>)<31:0>)
                 );!END IF TMPDOUBLEREG<1>
                (IF TMPIREG<0> EQL 1=>
                  (TMPIREG<0:15>←(MINUS TMPIREG<0:15>)<15:0>)
                )NEXT !END IF TMPIREG<0> EQL 1
                (DECODE ((TMPIREG<0:15> EQL 0) OR (TMPIREG<0:15> LSS TMPDOUBLEREG<1:16>))=>    !OVERFLOW?
                  ( (TMPOREG<0:15>←(TMPDOUBLEREG<1:32> / TMPIREG<0:15>)<15:0>)NEXT
                    (TMPIREG<0:15>←(TMPDOUBLEREG<1:32> MINUS (TMPOREG<0:15> * TMPIREG<0:15>)<31:0>)<15:0>)NEXT
                    (IF TMPOSIGN<0> EQL 1=>
                      (TMPOREG<0:15>←(MINUS TMPOREG<0:15>)<15:0>)
                    );!END IF TMPOSIGN<0> EQL 1
                    (IF TMPISIGN<0> EQL 1=>
                      (TMPIREG<0:15>←(MINUS TMPIREG<0:15>)<15:0>)
                    )NEXT !END IF TMPISIGN<0> EQL 1
                    (AC0<0:15>←TMPIREG<0:15>);
                    (AC1<0:15>←TMPOREG<0:15>);
                    (CARRY<0>←0)
                  );
                  ( (OVF<0>←1);
                    (CARRY<0>←1)
                  )
                );!END DECODE ((TMPIREG<0:15> EQL 0) OR (TMPIREG<0:15> LSS TMPDOUBLEREG<1:32>))
              )
            ););!END DECODE MDR<9>
            (INCR.PC)
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !end if
!END PART 2 OF 4 SINGLE-WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
!START PART 3 OF 4 SINGLE-WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
        (IF((MDR<5:7>@MDR<10:15>) EQL '100000001) AND (MDR<8:9> NEQ '11))=>
          (
            (IF MDR<8:9> EQL '00=>
XXOR:=        (AC[MDR<3:4>]<0:15>←AC[MDR<3:4>]<0:15> XOR AC0<0:15>)        !XOR  Exclusive OR
            );!END IF MDR<8:9> EQL '00
            (IF MDR<8:9> EQL '01=>
IOR:=         (AC[MDR<3:4>]<0:15>←AC[MDR<3:4>]<0:15> OR AC0<0:15>)         !IOR  Inclusize OR
            );!END IF MDR<8:9> EQL '01
            (IF MDR<8:9> EQL '10=>
DEC:=         (AC[MDR<3:4>]<0:15>←(AC[MDR<3:4>]<0:15> - 1)<15:0>)          !DEC  Decrement accumulator
            );!END IF MDR<8:9> EQL '10
            (INCR.PC)
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !end if
!END PART 3 OF 4 SINGLE-WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
!START PART 4 OF 4 SINGLE-WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
        (IF((MDR<5:7>@MDR<10:15>) EQL '110000001)=>
          (
            (DECODE MDR<8:9>=>
UMPY:=        ( (TMPDOUBLEREG<1:32>←AC[MDR<3:4>]<0:15> * AC1<0:15>)NEXT    !UMPY   UNSigned multiply
                (AC0<0:15>←TMPDOUBLEREG<1:16>);
                (AC1<0:15>←TMPDOUBLEREG<17:32>)
              );
UDVD:=        ( (TMPDOUBLEREG<1:32>←AC0<0:15>@AC1<0:15>);                 !UDVD   UNSigned divide
                (TMPIREG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
                (DECODE ((TMPIREG<0:15> EQL 0) OR (TMPIREG<0:15> LSS TMPDOUBLEREG<1:16>))=>    !OVERFLOW?
                  ( (TMPOREG<0:15>←(TMPDOUBLEREG<1:32> / TMPIREG<0:15>)<15:0>)NEXT
        •           (TMPIREG<0:15>←(TMPDOUBLEREG<1:32> MINUS (TMPOREG<0:15> * TMPIREG<0:15>)<31:0>)<15:0>)NEXT
                    (AC0<0:15>←TMPIREG<0:15>);
                    (AC1<0:15>←TMPOREG<0:15>);
                    (CARRY<0>←0)
                  );
                  ( (OVF<0>←1);
                    (CARRY<0>←1)
                  )
                );!END DECODE ((TMPIREG<0:15> EQL 0) OR (TMPIREG<0:15> LSS TMPDOUBLEREG<1:16>))
              );
UDVI:=        ( (TMPDOUBLEREG<1:16>←AC1<0:15>);                           !UDVI   UNSigned divide INTEGER
                (TMPIREG<0:15>←AC[MDR<3:4>]<0:15>)NEXT
                (DECODE (TMPIREG<0:15> EQL 0)=>      !OVERFLOW?
                  ( (TMPOREG<0:15>←(TMPDOUBLEREG<1:16> / TMPIREG<0:15>)<15:0>)NEXT
                    (TMPIREG<0:15>←(TMPDOUBLEREG<1:16> MINUS (TMPOREG<0:15> * TMPIREG<0:15>)<31:0>)<15:0>)NEXT
                    (AC0<0:15>←TMPIREG<0:15>);
                    (AC1<0:15>←TMPOREG<0:15>);
                    (CARRY<0>←0)
                  );
                  ( (OVF<0>←1);
                    (CARRY<0>←1)
                  )
                );!END DECODE ((TMPIREG<0:15> EQL 0) OR (TMPIREG<0:15> LSS TMPDOUBLEREG<1:16>))
              );
UMPA:=        ( (TMPDOUBLEREG<1:32>←((AC[MDR<3:4>]<0:15> * AC1<0:15>)<31:0> + AC0<0:15>)<31:0>)NEXT   !UMPA   UNSigned multiply ADD
                (AC0<0:15>←TMPDOUBLEREG<1:16>);
                (AC1<0:15>←TMPDOUBLEREG<17:32>)
              )
            );!END DECODE MDR<8:9>
            (INCR.PC)
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !end if
!END PART 4 OF 4 SINGLE-WORD INSTRUCTIONS WITH ACCUMULATOR (IN BITS 3-4)
!START PART 1 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
        (IF((MDR<3:7>@MDR<10:15>) EQL '00000000001)=>)
          (
            (DECODE MDR<8:9>=>)
CLEM:=        ( (EM<0>←0);                                          !CLEM   CLEAR EXPANDED MEMORY FLAG
                (INCR.PC)
                       );
STEM:=        ( (EM<0>←1);                                          !STEM   SET EXPANDED MEMORY FLAG
                (INCR.PC)
              );
PRT:=         ( (TMPADR<0:15>←SP<0:15>)NEXT                         !PRT  POP AND RETURN
                (SP<0:15>←(SP<0:15> + 1)<15:0>)NEXT
                (READ.Memory)NEXT
                (PC<0:15>←TMPMDR<0:15>)
              );
RTFNI:=      ( (TMPADR<0:15>←SP<0:15>);                       !RTFNI  RETURN FROM NESTED INTERUPT
                (SP<0:15>←(SP<0:15> + 1)<15:0>)NEXT
                (READ.Memory)NEXT
                (TMPADR<0:15>←5); INTMSK ← TMPMDR NEXT
                (WRITE.Memory)NEXT
                (TMPADR<0:15>←SP<0:15>);
                (SP<0:15>←(SP<0:15> + 1)<15:0>)NEXT
                (READ.Memory)NEXT
                (PC<0:15>←TMPMDR<0:15>)
              )
            )!END DECODE MDR<8:9>
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !end if
!END PART 1 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
!START PART 2 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
        (IF((MDR<3:15> EQL '0010011000001) OR (MDR<3:15> EQL '0011100000001))=>)
          (
            (IF MDR<3:15> EQL '0010011000001=>)
DSPD:=        ( (CPDATA<0:15>←AC1<0:15>));                          !DSPD   Display data in control pannel
                (INCR.PC)
              )
            );!END IF MDR<3:15> EQL '0010011000001
            (IF MDR<3:15> EQL '0011100000001=>)
TCO:=         ( (DECODE OVF<0>=>)                                   !TCO  TEST AND CLEAR OVERFLOW
                  ( (PC<0:15>←(PC<0:15> + 2)<15:0>)
                  );
                  ( (OVF<0>←0);
                    (INCR.PC)
                  )
                )!END DECODE OVF<0>
              )
            )!END IF MDR<3:15> EQL '0011100000001
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !end if
!END PART 2 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
!START PART 3 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
        (IF((MDR<3:8>@MDR<10:15>) EQL '001111000001)=>)
          (
            (DECODE MDR<9>=>)
STISZ:=       ( (TMPADR<0:15>←SP<0:15>)NEXT                   !STISZ  Increment top element of stack, skip if zero
                (READ.Memory)NEXT
                (TMPMDR<0:15>←(TMPMDR<0:15> + 1)<15:0>)NEXT
                (WRITE.Memory);
                (DECODE (TMPMDR<0:15> EQL 0)=>)
                  ( (INCR.PC)
                  );
                  ( (PC<0:15>←(PC<0:15> + 2)<15:0>)
                  )
                )!END DECODE (TMPMDR<0:15> EQL 0)
              );
PST:=         ( (TMPADR<0:15>←(SP<0:15> - 1)<15:0>)NEXT        !PST  PUSH STATUS ONTO STACK
                (SP<0:15>←TMPADR<0:15>);
                (TMPMDR<0:15>←STATUS<0:15>)NEXT
                (WRITE.Memory);
                (IF TMPADR<0:15> LSS #420 =>)     !Stack overflow?
                  ( (ION<0>←0);
                    (TMPMDR<0:15>←PC<0:15>);
                    (TMPADR<0:15>←#44)NEXT
                    (WRITE.Memory);
                    (TMPADR<0:15>←#45)NEXT
```

```
                        (READ.Memory)NEXT
                        (PC<0:15>←TMPMDR<0:15>)NEXT
                        (BAILOUT INPUT.OUTPUT)
                      )
                   )NEXT !END IF TMPADR<0:15> LSS #420
                   (INCR.PC)
                 )
              ))END DECODE MDR<9>
              NEXT BAILOUT INPUT.OUTPUT
            )
          ) NEXT !end if
!END PART 3 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
!START PART 4 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
          (IF((MDR<3>@MDR<5:15>) EQL '110011000001)=>
            (
              (DECODE MDR<4>=>)
STIBN:=         ( (IBN<0>←1)                    !STIBN  Set interupt branch/nest flag
                ));
CLIBN:=         ( (IBN<0>←0)                    !CLIBN  Clear interupt branch/nest flag
                )
              ));!END DECODE MDR<4>
              (INCR.PC)
              NEXT BAILOUT INPUT.OUTPUT
            )
          ) NEXT !end if
!END PART 4 OF 4 SINGLE WORD INSTRUCTIONS WITH NO ARGUMENTS
          (IF(MDR<3:15> EQL '0011101000001)=>
PJS:=       ( (TMPMDR<0:15>←(TMPADR<0:15> + 2)<15:0>)NEXT   !PJS  Push and jump to subroutine
              (TMPREG<0:15>←(TMPADR<0:15> + 1)<15:0>)NEXT


              (TMPADR<0:15>←(SP<0:15> - 1)<15:0>)NEXT
              (SP<0:15>←TMPADR<0:15>);
              (WRITE.Memory)NEXT
              (IF TMPADR<0:15> LSS #420 =>    !Stack overflow?
                ( (ION<0>←0);
                  (TMPMDR<0:15>←PC<0:15>);
                  (TMPADR<0:15>←#44)NEXT
                  (WRITE.Memory)NEXT
                  (TMPADR<0:15>←#45)NEXT
                  (READ.Memory)NEXT
                  (PC<0:15>←TMPMDR<0:15>)NEXT
                  (BAILOUT INPUT.OUTPUT)
                )
              )NEXT !END IF TMPADR<0:15> LSS #420
              (TMPADR<0:15>←TMPREG<0:15>)NEXT
              (READ.Memory)NEXT
              (PC<0:15>←TMPMDR<0:15>)
              NEXT BAILOUT INPUT.OUTPUT
            )
          ) NEXT !end if



          (IF(MDR<3:15> EQL '0110011000001)=>
FS:=        ( (TMPADR<0:15>←(TMPADR<0:15> + 1)<15:0>)NEXT   !FS   File search
              (READ.Memory)NEXT
              (TMP0REG<0:15>←TMPMDR<0:15>);
              (TMP1REG<0:15>←AC2<0:15>)NEXT
LOOP2:=       (                                             !START LOOP2
                (IF (AC3<0:15> EQL TMP1REG<0:15>)=>
                  ( (PC<0:15>←(PC<0:15> + 2)<15:0>);
                    (BAILOUT INPUT.OUTPUT)
                  )
                )NEXT !END IF (AC3<0:15> EQL TMP1REG<0:15>)
                (TMP1REG<0:15>←(TMP1REG<0:15> + 1)<15:0>)NEXT
                (TMPADR<0:15>←TMP1REG<0:15>)NEXT
                (R.DATA)NEXT
                (TMPMDR<0:15>←TMPMDR<0:15> AND TMP0REG<0:15>)NEXT
                (IF ((AC0<0:15> LEQ TMPMDR<0:15>) AND (TMPMDR<0:15> LEQ AC1<0:15>))=>
                  ( (PC<0:15>←(PC<0:15> + 3)<15:0>)NEXT
                    (BAILOUT INPUT.OUTPUT)
                  )
                )NEXT !END IF ((AC0<0:15> LEQ TMPMDR<0:15>) AND (TMPMDR<0:15> LEQ AC1<0:15>))=>
                (AC2<0:15>←TMP1REG<0:15>)NEXT
                (LOOP2)
              )!END LOOP2
              NEXT BAILOUT INPUT.OUTPUT
            )
          ) NEXT !end if
!START PART 1 of 3  CODE-77 IO WITHOUT ACCUMULATOR
          (IF((MDR<3:4>@MDR<10:15> EQL '00111111)AND((MDR<5:9> GEQ '1)AND(MDR<5:9> LEQ '11)))=>
            (
              (IF (MDR<5:9> EQL '1)=>
INTEN:=         ( (ION<0>←'1);                              !INTEN  Interrupt enable
                (INCR.PC)
              )
            ));!END IF (MDR<5:9> EQL '1)=>
            (IF (MDR<5:9> EQL '10)=>
INTDS:=       ( (ION<0>←'0);                          !INTDS  Interrupt disable
                (INCR.PC)
              )
            ));!END IF (MDR<5:9> EQL '1)=>
            (IF (MDR<5:9> EQL '11)=>
CLPD:=        ( (CPDATA<0:15>←'0);                            !CLRD  Clear control panel display
                (INCR.PC)
              )
            )!END IF (MDR<5:9> EQL '1)=>
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !end of if
!END  PART 1 of 3  CODE-77 IO WITHOUT ACCUMULATOR


!START  PART 2 of 3  CODE-77 IO WITHOUT ACCUMULATOR
        (IF((MDR<3:15> EQL '0010110111111) OR (MDR<3:15> EQL '0011000111111))=>
          (
            (IF (MDR<5:9> EQL '10110)=>
IORST:=       ( (ION<0>←'0);                          !IORST  I/O Reset
                (INCR.PC)
              )
            ));!END IF (MDR<5:9> EQL '10110)
            (IF (MDR<5:9> EQL '11000)=>
HALT:=        ( (INCR.PC) NEXT                  !HALT
                (STOP)
              )
            )!END IF (MDR<5:9> EQL '11000)
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !end if
!END  PART 2 of 3  CODE-77 IO WITHOUT ACCUMULATOR


!START  PART 3 of 3  CODE-77 IO WITHOUT ACCUMULATOR
        (IF(MDR<3:7>@MDR<10:15> EQL '00111111111)=>
          (
            (DECODE MDR<8:9> =>
SKPBN.CPU:=   ( (DECODE (ION<0> EQL '1)=>                 !SKPBN CPU  Skip if interrupt enabled
                  ( (INCR.PC)
                  );
                  ( (PC<0:15>←(PC<0:15> + 2)<15:0>)
                  )
                )!END DECODE (ION<0> EQL '1)
              );
SKPBZ.CPU:=   ( (DECODE (ION<0> EQL '0)=>                 !SKPBZ CPU  Skip if interrupt disabled
                  ( (INCR.PC)
                  );
                  ( (PC<0:15>←(PC<0:15> + 2)<15:0>)
                  )
                )!END DECODE (ION<0> EQL '1)
              );
SKPDN.CPU:=   ( (INCR.PC)          !SKPDN CPU  Skip on power failing
              );
SKPDZ.CPU:=   ( (PC<0:15>←(PC<0:15> + 2)<15:0>)   !SKPDZ CPU  Skip on power OK
              )
            )!END DECODE MDR<8:9>
            NEXT BAILOUT INPUT.OUTPUT
          )
        ) NEXT !END OF IF
!END   PART 3 of 3  CODE-77 IO WITHOUT ACCUMULATOR


!START CODE 77 I/O WITH ACCUMULATOR

        (IF MDR<10:15> EQL #77 =>
              (DECODE MDR<5:7>=>
                        (BAILOUT INPUT.OUTPUT);
        READS:=         (AC[MDR<3:4>]←SWITCH);
                        (BAILOUT INPUT.OUTPUT);
        INTA:=          (AC[MDR<3:4>]←DEV.NUMBER);
        MSKO:=          (INTMSK ← AC[MDR<3:4>]);
                        (BAILOUT INPUT.OUTPUT);
                        (BAILOUT INPUT.OUTPUT);
                        (BAILOUT INPUT.OUTPUT)
              )NEXT
              PC ← (PC+1)<15:0>;
              (DECODE MDR<8:9> =>
                        (BAILOUT INPUT.OUTPUT);
                        ION ← 1;
                        ION ← 0;
                        LPDATA ← 0
                    )NEXT
                    BAILOUT INPUT.OUTPUT
        ); !END CODE 77 I/O WITH ACCUMULATOR
!**********FLOATING POINT ISPL (IMPLEMENTED AS FIXED POINT)***********
!1602 optional extended instruction set - floating point instructions
!************** WDIS  7/9/77 ***************************************


        (if((mdr<5:7> eql '100) and (mdr<10:15> eql 0)=>
          (!start floating point arithmetic
            tmpadr<0:15>←(tmpadr<0:15>+1)<15:0> next
            READ.Memory next
            tmpadr<0:15>←tmpmdr<0:15> next
            R.DATA next
            td1<0:15>←tmpmdr<0:15> next
            tmpadr<0:15>←(tmpadr<0:15>+1)<15:0> next
            R.DATA next
            td1<16:31>←tmpmdr<0:15> next
            (decode mdr<8:8>=>
FAD:=         ((tmpdoublereg<0:32>←((td1<0:31> + ac0<0:15>@ac1<0:15>)<32:0>));
FSB:=         ((tmpdoublereg<0:32>←(ac0<0:15>@ac1<0:15> MINUS td1<0:31>)<32:0>));
FMP:=           BEGIN
                td0<0:31> ← ac0<0:15>@ac1<0:15> next
                sign ← td0<0> XOR td1<0> next
                (IF td0<0> => td0<0:31> ← (MINUS td0<0:31>)<31:0>);
                (IF td1<0> => td1<0:31> ← (MINUS td1<0:31>)<31:0>) NEXT
                tmpdoublereg<0:32> ← ((TD0 * TD1) !SR0 16)<31:0> NEXT
                (IF sign => tmpdoublereg<0:32> ← (MINUS tmpdoublereg)<31:0>)
                END;

FDV:=           BEGIN
                (if td1<0:31> eql 0 =>
                        (ovf<0>←1; INCR.PC next
                        bailout input.output) !divide by 0
                ) next !end if
                td0<0:31>← ac0<0:15>@ac1<0:15> next
                sign ← td0<0> XOR td1<0> NEXT
                (IF td0<0> => td0 ← (MINUS td0<0:31>)<31:0>);
                (IF td1<0> => td1 ← (MINUS td1<0:31>)<31:0>) NEXT
                tmpdoublereg ← ((td0@0<15:0>) / td1)<31:0> NEXT
                (IF sign => tmpdoublereg ← (MINUS tmpdoublereg<1:32>)<31:0>)
                END
         ) NEXT
           ac0<0:15>←tmpdoublereg<1:16>;
           ac1<0:15>←tmpdoublereg<17:32> next
           (decode mdr<3:4>=>
             (NOP);                                  !no operation
             (if(tmpdoublereg<1> eql 0)=>(INCR.PC)); !skip on positive
             (if(tmpdoublereg<1> eql 1)=>(INCR.PC)); !skip on negative
             (NOP)                                   !normalize(not implemented)
           ) next !end decode
           pc<0:15>←(pc+2)<15:0> next
           bailout input.output
          )!end floating point arithmetic
        ) next !end if



        (if(mdr<5:7> eql '101) and (mdr<10:15> eql 0)=>
          (!start floating point conversion
            (decode mdr<8:8>=>
FLO:=         (NOP);                        !float num in ac0,ac1(not implemented)
FIX:=         (NOP);                        !fix   "      "  "  "        "
FNM:=         (NOP);                        !normalize num in " "  "      "
              (NOP)                         !unused instr
            ) next !end decode
            (decode mdr<3:4>=>
              (NOP);                        !no operation
              (if(ac0<0> eql 0)=>(INCR.PC)); !skip on positive
              (if(ac0<0> eql 1)=>(INCR.PC)); !skip on negative
              (NOP)                         !normalize(not implemented)
            ) next !end decode
            INCR.PC NEXT
            bailout input.output
          )!end floating point conversion
        ) next !end if


 !****************RMU Resource Management Unit (ROLM 1666)********

        (if(mdr<3:7> eql '00111) and (mdr<10:15> eql 0)=>
          begin !PMU decode WD15 7/13/77
            INCR.PC NEXT
            (decode mdr<8:9>=>
              nop!                   !unused instr
              RLAF;
              FIX;
              ECALL
            ) next !end decode
            bailout input.output
          end !PMU decode
        ) next !end if


!*********************Catch Illegal Instructions*******************

        (if (MDR<6:7> eql '10) and (MDR<10:15> eql 0)=> illegal) next
                !DOA, Device Address(DA)=00 and DOC, DA=00

        (if (MDR<3:7> eql '00111) and (MDR<10:15> eql 0)=> illegal) next
                !SKPBN 00, SKPBZ 00, SKPDN 00 and SKPDZ 00

        (if (MDR<3:4> neq 0) and (MDR<5:7> eql '111)=> illegal) next
                !IOSKP group, MDR<3:4> = 01, 10 or 11

        (if (MDR<3> eql 1) and (MDR<5:7> eql 0)=> illegal) next
                !NIO group, MDR<3:4> = 10 or 11

!       (if (MDP<5:6> eql '10) and (MDP<10:15> eql 0)=> illegal) next
                !DIC, DA=00 and DOB, DA=00
                !Used for floating point option

!       (if (MDR<3:7> eql 0) and (MDR<10:15> eql 0)=> illegal) next
                !NIO, DA=00
                !Used for floating point option


!***************Input-Output Instructions*********************
        (PC←(PC+1)<15:0>  NEXT
DIO:=   (DECODE MDR<5:7> =>
                NOP;            !NIO
                INDEV;          !DIA
                OUTDEV;         !DOA
                INDEV;          !DIB
                OUTDEV;         !DOB
                INDEV;          !DIC
                OUTDEV;         !DOC
                NOP             !SKP INSTRUCTIONS
        )NEXT
        BAILOUT INPUT.OUTPUT)


));END INPUT.OUTPUT


!       S T O R E . A C C U M U L A T O R      Uses Variables  TMPMDR<0:15>, TMPADR<0:15>, AC[0:3]<0:15>,
!                                                              MDR<3:4>, PC<0:15>.
!       S T O R E . A C C U M U L A T O R      Uses Routines   WRITE.Memory
!       S T O R E . A C C U M U L A T O R

STORE.ACCUMULATOR:=          !>>> Store the contents in the specified Accumulator in the
                             !>>>    effective Memory location (already in TMPADR) and increment the PC.

(!START STORE.ACCUMULATOR



    (TMPMDR<0:15>←AC[MDR<3:4>]<0:15>)NEXT
    (W.DATA);
    (INCR.PC)

));END STORE.ACCUMULATOR
!       L O A D . A C C U M U L A T O R        Uses Variables  AC[0:3]<0:15>, MDR<3:4>,
!                                                              TMPMDR<0:15>, TMPADR<0:15>.
!       L O A D . A C C U M U L A T O R        Uses Routines   READ.Memory
!       L O A D . A C C U M U L A T O R

LOAD.ACCUMULATOR:=           !>>> Load data from effective address data (in TMPMDR) into specified
                             !>>>    accumulator and increment PC by 1.

(!START LOAD.ACCUMULATOR

    (AC[MDR<3:4>]<0:15>←TMPMDR<0:15>);
    (INCR.PC)

));!END LOAD.ACCUMULATOR
!       N O A C . E F F E C T I V E . A D D R E S S     Uses Variables  MDR<3:4>, PC<0:15>, TMPADR<0:15>,
!                                                                        TMPMDR<0:15>.
!       N O A C . E F F E C T I V E . A D D R E S S     Uses Routines   READ.Memory, WRITE.Memory.
!       N O A C . E F F E C T I V E . A D D R E S S


NOAC.EFFECTIVE.ADDRESS:=   !>>> Decode op-code of NO Accumulator Effective Address format
                           !>>>     instruction, then increment PC appropriately.

(!START NOAC.EFFECTIVE.ADDRESS

    (DECODE MDR<3:4> =>)                          !Decode op-code
JMP:=    (    (PC<0:15>+TMPADR<0:15>)
         );                                        !00=JMP  Program Counter-Effective Address
JSR:=    (    (AC[3]<0:15>+(PC<0:15> + 1)<15:0>)) NEXT
             *(PC<0:15>+TMPADR<0:15>))
         );                                        !01=JSR  Jump to subroutine saving PC+1.
ISZ:=    (
              (R.DATA)NEXT
              (TMPMDR<0:15>+(TMPMDR<0:15> + 1)<15:0>)NEXT
              (W.DATA));
              (DECODE (TMPMDR<0:15> EQL 0) =>)Skip one if zero
                  (INCR.PC);
                  (PC<0:15>+(PC<0:15> + 2)<15:0>))
              )!END DECODE(TMPMDR<0:15> EQL 0)
         );                                        !10=ISZ  Increment the Effective Address contents and Skip if Zero
DSZ:=    (
              (R.DATA)NEXT
              (TMPMDR<0:15>+(TMPMDR<0:15> - 1)<15:0>)NEXT
              (W.DATA);
              (DECODE (TMPMDR<0:15> EQL 0) =>)Skip one if zero
                  (INCR.PC);
                  (PC<0:15>+(PC<0:15> + 2)<15:0>))
              )!END DECODE (TMPMDR<0:15> EQL 0)
         )                                         !11=DSZ  Decrement the Effective Address contents and Skip if Zero
    )!END DECODE MDR<3:4>

));!END NOAC.EFFECTIVE.ADDRESS
IEXEC:= BEGIN
        (TMPADR<0:15>+PC<0:15>)NEXT
        (READ.Memory)NEXT
        (MDR<0:15>+TMPMDR<0:15>)NEXT
        (DECODE MDR<0> =>)                                   !1=Arithmetic or Logic operation
            (
              (DECODE MDR<1:2> =>)
                  (   (ADR.SETUP)NEXT          !Get the effective address in TMPADR
                      (NOAC.EFFECTIVE.ADDRESS)
                  );                                        !00= No Accumulator-Effective Address format
LDA:=             (   (ADR.SETUP)NEXT          !Get the effective address in TMPADR
                      (R.DATA)NEXT             !Get data in TMPMDR
                      (LOAD.ACCUMULATOR)
                  );                                        !01= part of One Accumulator-Effective Address format
STA:=             (   (ADR.SETUP)NEXT          !Get the effective address in TMPADR
                      (STORE.ACCUMULATOR)
                  );                                        !10= part of One Accumulator-Effective Address format
                  (
                      (INPUT.OUTPUT)                        !11= I/O format
                                                            !   (This format has numerous extended (HACKED) instructions.)
                  )
              )!END DECODE MDR<1:2>
            );
            (ARITHMETIC.or.LOGIC)
        )NEXT !END DECODE MDR<0>
        MSI + 0 ; MSD + 0
        END;
INTERRUPT:=     BEGIN
                DECODE IBN =>
                    (ION + 0;
                    MEMORY[0] + PC NEXT          !ORDINARY INTERRUPT
                    TMPADR + "0001  NEXT                 !INDIRECT THRU LOC 1
                    ADR.FETCH NEXT
                    PC + TMPADR);

                    !BRANCH SEQUENCE INTERRUPT
                    (TMPREG + MEMORY[ (DEV.NUMBER+MEMORY[1])<15:0>] NEXT
                    (DECODE TMPREG<0> =>
                        !SIMPLE
                        (MEMORY[0] + PC NEXT
                        MEMORY[4] + AC3; ION + 0; PC + TMPREG<1:15>
                        );

                        !BRANCH AND NEST
                        (TMPMDR + PC NEXT
                        PUSH.STACK NEXT
                        TMPMOR ← MEMORY[S] NEXT !GET CURRENT MASK
                        PUSH.STACK NEXT
                        INTMSK ← MEMORY[(TMPOPEG-1)<14:0>] NEXT
                        MEMORY[S] ← INTMSK NEXT
                        PC ← TMPOPEG<1:15>
                        )
                    )
                )
            END

!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>END ROUTINES

EPALCED

!       I N S T R U C T I O N . D E C O D E
!       I N S T R U C T I O N . D E C O D E
!       I N S T R U C T I O N . D E C O D E


INSTPUCTION.DECODE:=        !>>> Decode the next user instruction.

        BEGIN
!       (status<0:15>←'0000001111111100)next
LOOP3:=         BEGIN
                (IF (ION NEQ 0) AND ((NOT INTMSK AND DEV.IBIT) NEQ 0) =>
                                INTEPRUPT) NEXT
                IEXEC NEXT
                LOOP3
                END
        END

)!END ROLM 1602
```

5. AN/UYK-20 ISPL Description

```
! T H E   A N / U Y K - 2 0   I S P
! ---------------------------------
!
!
!
! The AN/UYK-20 ISP is based on the information contained in
! the SPERRY UNIVAC document "AN/UYK-20  TECHNICAL DESCRIPTION".
!
!The ISP description includes all the instructions listed in the
!above document except for the following:
!
!        1) The Trigonometric and Hyperbolic Functions (OPCODE #37)
!        2) The Floating Point Instructions (OPCODES #50 to #53)
!        3) The Double Multiply and Divide Instructions (OPS #56 and #57)
!        4) The Square Root Instruction (OPCODE #4, M-desig 0)
!        5) The I/O Instructions (OPCODES #70 to #77)
!
!The Interrupt system and the IOC channels are partially implemented.
!In each case , the required state is defined.The only interrupt
!implemented,however, is the Class II-Priority 1 CP Instruction Fault.
!This is generated when execution of an unassigned opcode is attempted.
!The un-implemented instructions (except for floating-point) are treated
!as if they were unassigned opcodes. The service routine for this
!interrupt is merely a trap through location 000000, after which
!the machine is halted.

!A pseudo floating-point instruction set is substituted for the "genuine"
!one. A simple F.P. format is assumed, whereby each F.P. number occupies
!2 consecutive words (the more-significant being on an even boundary), with
!an implicit binary point between the two words. This allows the use of
!the integer arithmetic of the PDP-10 (on which the simulator runs),
!instead of its floating-point, which is noncompatible with that of
!the UYK-20.
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!Send complaints and/or comments to Karem Sakallah @CMUD.
```

```
UYK20 :=
(DECLARE

!       Main memory
!       -----------

        MP[0:#177777]<15:0>,               ! 64 K words

!       Non-destructive read-only memory for bootstrap loading
!       ------------------------------------------------------

        NDRO[0:#77,#300:#477]<15:0>,

!       Addressable register definitions
!       --------------------------------

        R0[0:#17]<15:0>,                   !General registers
        R1[0:#17]<15:0>,                   !Optional second set

        PAR[0:#77]<15:0>,                  !Page address registers

        RTC<31:0>,                         !Real time clock

        MON<15:0>,                         !Monitor clock

        BRKPNT<15:0>,                      !Breakpoint register

!       Processor state registers
!       -------------------------

        P<15:0>,                           !Program address register

        SR1<15:0>,                         !Status register 1
        RSELCT<> := SR1<14>,               !General register select bit
        MSELCT<> := SR1<12>,               !Main or ndro memory select bit
        CARRY<> := SR1<11>,                !Carry Bit
        OVRFLW<> := SR1<10>,               !Overflow bit
        CCDES<1:0> := SR1<9:8>,            !Condition code designator
        OVRFWI<> := SR1<7>,                !Enable overflow interupt
        FLPTRI<> := SR1<6>,                !Enable floating point round
        CLASSI<> := SR1<3>,                !Enable class I interupts
        CLASII<> := SR1<2>,                !Enable class II interupts
        CLAIII<> := SR1<1>,                !Enable class III interupts
        DMA<>    := SR1<0>,                !Enable dma

        SR2<15:0>,                         !Status register 2
        ICB16<1:0>  := SR2<15:14>,         !Indirect control bits for register 16
        ICB14<1:0>  := SR2<13:12>,         !Indirect control bits for register 14
        ICB12<1:0>  := SR2<11:10>,         !Indirect control bits for register 12
        ICB10<1:0>  := SR2<9:8>,           !Indirect control bits for register 10
        INPICD<7:0> := SR2<7:0>,           !Interupt code

!       Instruction Register
!       --------------------

        IRTEMP<15:0>,                      ! Temporary Instruction Register
        IR<15:0> := IRTEMP<15:0>,          ! Instruction Register
        OPCODE<5:0> := IRTEMP<15:10>,      !Opcode
        FCODE<1:0> := IRTEMP<9:8>,         !Format code
        AREG<3:0> := IRTEMP<7:4>,          !A register designator
        MREG<3:0> := IRTEMP<3:0>,          !M register designator
        SCODE<> := IRTEMP<7>,              !Sign designator for local jumps
        DCODE<6:0> := IRTEMP<6:0>,         !Displacement designator for local jumps

!       Internal registers
!       ------------------

        MICPOP<15:0>,                      !Microprogram counter
        I1<3:0>,                           !Interupt code 1 store
        I2<6:0>,                           !Interupt code 2 store
```

```
!       Indicators and switches from the control panel
!

        STP<>:                        !Program stop indicator
                                      !Lights to indicate program stop condition
        STOP1<>:                      !Program stop switch number 1
        STOP2<>:                      !Program stop switch number 2
        BST<>:                        !Bootstrap program select switch
        LDST<>:                       !Load switch
                                      !Causes a master clear followed by a load
        MSCLR<>:                      !Master clear switch
        RUN<>:                        !Run indicator.  Lights when in run mode
        POT<>:                        !Power out of tolerance indicator
        PGFALT<>:                     !Program fault indicator
        PGFLTC<>:                     !Program fault clear
        BRPTRD<>                      !Breakpoint read enable switch
        BRPTWT<>:                     !Breakpoint write enable switch
        DJ<>:                         !Diagnostic jump switch

!       I/O REGISTER DEFINITIONS
!       ------------------------

        IOCC[0:#277]<15:0>:           !Ioc channel control memory
                                      !12 registers for each of 16 channels
        INPUT[0:#17]<15:0>:           !Input ports
        OUTPUT[0:#17]<15:0>:          !Output ports

        EIE<15:0>:                    !External interrupt enables
        EXT<15:0>:                    !External interrupt channel number
          PBITS[15:0]<> := EXT<15:0>;
        EI<15:0>:                     !External interrupt lines
          PBITS[15:0]<> := EI<15:0>;
        DATREQ<31:0>:                 !Data request lines
          QBITS[31:0]<> := DATREQ<31:0>;
        CHAIN<31:0>:                  !I/o program chain instruction request lines
          SBITS[31:0]<> := CHAIN<31:0>;
```

[illegible]

```
[illegible]

   [illegible]                              [illegible]
   [illegible]
   [illegible]                              [illegible]
   [illegible]                              [illegible]
OPR<15:0>,                              !Operand from memory
OPR1<15:0>,                             !Second half of double length operand from memory
[illegible]<15:0>,                      !Temporary register
[illegible]<15:0>,                      !another one
IW1<15:0>,                              !Indirect word 1
IW2<15:0>,                              !Indirect word 2
 [illegible]<3:0> := IW1<3:0>,          !X Register designator
   [illegible]<1:0> := IW1<13:12>,
BYTF<>,                                 !Byte operand flag
BYTSEL<>,                               !1=/0= byte selector

MEMADD<15:0>,                           !Holds memory address
MEMVAL<15:0>,                           !Holds value returned from memory
TEMPD1<32:0>,                           !Double length operand created from Ra and Ra+1
TEMPD2<32:0>,                           ! Another one
TEMPD3<64:0>,                           !For pseudo floating point instructions
DQUOT<63:0>,                            !double word quotient
TEMP<15:0>,                             !Temporary register
TMPBITS[15:0]<> := TEMP<15:0>,
STEMP<15:0>,                            !Another one
ONE<>,                                  !One bit temporary
DOS<>,                                  !another one
ADC<>,                                  !A third one
TMP1<1:0>,                              !2 bit temporary
TMP2<1:0>,                              !another one

CTEMP<15:0>,                            !Counting register for identifying IC channel
CLASS<1:0>,                             ! Interrupt class
RICL<>,
MEM<>,
MONINE<>,
RICOI<>,
STPLG<>,
IOL<>,
FAULT<>,
FOULC<>,
OUT<>,
INPLU<>,
INTFLG<>,                               ! Set when there is an interrupt
HONOR<>,                                ! Set when interrupt is honored
RLXFLG<>,                               !Remote Execute instruction flag
WHICHM<>,                               !Memory selector (MP or NDRO)
```

```
!       [illegible]
!

!       [illegible]TION
!       **********************

!       Read Rm
        MSEL :=
        ((DECODE PSELECT => OPRM<15:0> ← R0[MREG]<15:0>; OPRM<15:0> ← R1[MREG]<15:0>);
                OPRM<16> ← 0
        );

!       Read Ra
        ASELR:=
        ((DECODE PSELECT => OPRA<15:0> ← R0[AREG]<15:0>; OPRA<15:0> ← R1[AREG]<15:0>);
                OPRA<16> ← 0
        );

!       Read Rx
        XSEL :=
        ((DECODE PSELECT => OPRX<15:0> ← R0[XREG]<15:0>; OPRX<15:0> ← R1[XREG]<15:0>);
                OPRX<16> ← 0
        );

!       Write in Ra
        ASELW :=
        (DECODE PSELECT => R0[AREG] ← OPRA<15:0> ; R1[AREG] ← OPRA<15:0> );

!       Write double-word operand in Ra and Ra+1
        ASELDW :=
        (DECODE PSELECT =>
                (R0[AREG] ← TEMPD1<31:16> ; R0[(AREG OR '0001)] ← TEMPD1<15:0>);
                (R1[AREG] ← TEMPD1<31:16> ; R1[(AREG OR '0001)] ← TEMPD1<15:0>)
        );

!       Read double-word operand from Ra and Ra+1
        ASELDR :=
        (       ASELR
                 next   TEMPD1<32:16> ← OPRA
                next    (DECODE PSELECT =>
                      ·  TEMPD1<15:0> ← R0[(AREG OR '0001)] ;
                         TEMPD1<15:0> ← R1[(AREG OR '0001)]
                )
        );
```

```
!       UTILITIES / 2
!

!       Memory  Management and Selection
!       ****************
        MNGMNT :=
        (TEMP<15:10>←PAR[MEMADD<15:10>]<5:0>; TEMP<9:0>←MEMADD<9:0> next
         WHICHM ← (
                        (
                                (TEMP LEQ #77) OR
                                ((TEMP LEQ #477) AND (TEMP GEQ #300))
                        ) AND
                        (MSELECT EQL 0)
                )
        );

!       Memory Read
!       ***********************
        MEMREF:=        !Memory reference, address in MEMADD, data in MEMVAL
        (MNGMNT next
         (DECODE WHICHM =>
                (MEMVAL<15:0>←MP[TEMP]<15:0>;MEMVAL<16>←0);
                (MEMVAL<15:0>←NORO[TEMP]<15:0>;MEMVAL<16>←0)
         )
        );

!       Memory Write
!       ***********
        MEMOUT :=       !Single-word or byte memory write, address in MEMADD, data in OPRY
        (MNGMNT next
         (1): (WHICHM EQL 0) =>
                PAR[MEMADD<15:10>]<15> ← 1
                next    (DECODE BYTE =>
                                MP[TEMP] ← OPRY<15:0> ;
                                (DECODE BYTSEL =>
                                        MP[TEMP]<15:8> ← OPRY<7:0> ;
                                        MP[TEMP]<7:0> ← OPRY<7:0>
                                )
                        )
         )
        );

        MEMOTD :=       !Double Memory Write
        (       MEMOUT
                next    MEMADD ← (MEMADD OR 1)<15:0> ; OPRY<15:0> ← OPRYL
                next    MEMOUT
        );

!       Indirect memory addressing mode
!       *******************************
        INDRCT  :=      !leaves address of indirect operand in MEMADD
        (       MEMREF
                next    IW1 ← MEMVAL<15:0> ; MEMADD ← (MEMADD+1)<15:0>
                next    MEMREF
                next    IW2 ← MEMVAL<15:0>
                next    (DECODE JFIELD =>
                        TEMP ← 0 ;
                        (       XSEL
                                next    TEMP ← OPRX<15:0>
                        );
                        TEMP ← OPRM<15:0> ;
                        (       MREG ← (MREG+1)<3:0>
                                next    MSEL
                                next    TEMP ← OPRM<15:0>
                        )
                )
                next    (IF (IW1<14> EQL 1) =>  (       MEMADD ← (IW2+TEMP)<15:0>
                                next    INDRCT
                        )
                );
                (IF (IW1<14> NEQ 1) =>  (DECODE BYTE =>
                                MEMADD ← (IW2+TEMP)<15:0> ;
                                (MEMADD ← (IW2+(TEMP SR0 1))<15:0> ; BYTSEL ← TEMP<0>)
                        )
                )
         );
```

```
!       UTILITIES / 3
!       --------- ---

!       INSTRUCTION FORMATS
!       *******************

!       Register to Register
        RR :=
        (MSEL next
                OPRY ← OPRM
        ));

!       Register to Immediate
        RI :=           ! leaves address of operand in MEMADD
        (MSEL next
                MEMADD ← OPRM<15:0>
        ));

        RI1:=           ! leaves the operand in OPRY
        (RI next MEMREF next OPRY ← MEMVAL));

!       Register - Constant
        RK :=
        (MSEL;
                MEMADD ← PCOUNT next
                PCOUNT ← (PCOUNT+1)<15:0>;
                MEMREF next
                (IF (MREG EQL 0) => OPRY ← MEMVAL);
                (IF (MREG NEQ 0) => OPRY<15:0> ← (MEMVAL+OPRM)<15:0>; OPRY<16> ← 0)
        ));

!       Register Indexed
        INDX :=         !Indexing
        (DECODE         BYTE =>
                MEMADD ← (MEMVAL+OPRM)<15:0> ;
                (MEMADD ← (MEMVAL+(OPRM SR0 1))<15:0> ; BYTSEL ← OPRM<0>)
        ));

        RX :=           !leaves address of operand in MEMADD
        (MSEL;
                MEMADD ← PCOUNT next
                PCOUNT ← (PCOUNT+1)<15:0>;
                MEMREF next
                (IF (MREG EQL 0) => MEMADD ← MEMVAL<15:0> ; BYTSEL ← 0);
                (IF (MREG NEQ 0) =>
                        (IF (MREG NEQ #10) AND (MREG NEQ #12) AND (MREG NEQ #14) AND (MREG NEQ #16) => INDX);
                        (IF ((MREG EQL #10) OR (MREG EQL #12) OR (MREG EQL #14) OR (MREG EQL #16)) =>
                                (IF (MREG EQL #10) => ITEMP<1:0> ← ICB10);
                                (IF (MREG EQL #12) => ITEMP<1:0> ← ICB12);
                                (IF (MREG EQL #14) => ITEMP<1:0> ← ICB14);
                                (IF (MREG EQL #16) => ITEMP<1:0> ← ICB16) next
                                (DECODE ITEMP<1:0> =>
                                        INDX;
                                        INDX;
                                        (MEMADD ← MEMVAL<15:0> next INDRCT);
                                        (MEMADD ← (MEMVAL+OPRM)<15:0> next INDRCT)
                                )
                        )
                )
        ));

        RX1 :=          ! leaves operand in OPRY
        (RX next MEMREF next OPRY ← MEMVAL));
```

```
!       UTILITIES / 4
!        -------------

!       INSTRUCTION FORMAT DECODING
!       ***************************

        FDCODE :=
        (DECODE FCODE =>
                PR;
                RI1;
                RK;
                RX1
        );

!       Hi/Lo Byte selection
!       ********************

        BYTRD :=
        (       TEMP ← OPRY<15:0>
                next    OPRY<15:0> ← 0
                next    (DECODE BYTSEL =>
                        OPRY<7:0> ← TEMP<15:8> ;
                        OPRY<7:0> ← TEMP<7:0>
                )
        );

!       BYTE WRITE
!       **********

        BYTWT :=
        (       ASELR
                next    OPRY<7:0> ← OPRA<7:0>
                next    MEMOUT
        );

!       Read Double-word operand from Rm or memory
!       ******************************************

        DOBSEL :=
        ((IF (FCODE EQL 0) =>
                        MSEL next
                        OPRY ← OPRM next
                        (DECODE RSELCT => OPRY1 ← R0[(MREG OR '0001)]; OPRY1 ← R1[(MREG OR '0001)])
                );
                (IF (FCODE NEQ 0) =>
                        FOCODE next
                        MEMADD ← (MEMADD OR 1)<15:0> next
                        MEMREF next
                        OPRY1 ← MEMVAL<15:0>
                )
        );

!       CONDITION CODES
!       ***************

        CC :=                           !For single operands
        ((IF (OPRA<15:0> EQL 0) => CCDES ← 0);
                (IF (OPRA<15> EQL 1) => CCDES ← 3);
                (IF (OPRA<15> EQL 0) AND (OPRA<14:0> NEQ 0) => CCDES ← 1)
        );


        CCD :=                          !For double operands
        ((IF (TEMPD1<31:0> EQL 0) => CCDES ← 0);
                (IF (TEMPD1<31> EQL 1) => CCDES ← 3);
                (IF (TEMPD1<31> EQL 0) AND (TEMPD1<30:0> NEQ 0) => CCDES ← 1)
        );
```

```
!       UTILITIES / 5
!       ---- ---------

!       Interrupt Servicing
!       ******************

   INTPPT :=
        ((IF INTFLG =>
                (HONOP ← 0 next
                 (IF (CLASS EQL 1) AND CLASS1 => (HONOP←1;ITEMP←#20 )) ;
                 (IF (CLASS EQL 2) AND (I1 EQL 0)  => (HONOP←1;ITEMP←#10 )) ;    ! Unassigned ops
                 (IF (CLASS EQL 2) AND CLAS11 => (HONOP←1;ITEMP←#10 )) ;
                 (IF (CLASS EQL 3) AND CLA111 => (HONOP←1;ITEMP←0 )) next
                 (IF HONOP =>
                     (MEMADD ← (ITEMP+#110)<15:0>; OPRY<15:0> ← P next
                      MEMOUT next
                      MEMADD ← (MEMADD+1)<15:0>; OPRY<15:0> ← SR1 next
                      MEMOUT next
                      MEMADD ← (MEMADD+1)<15:0>; OPRY<15:0> ← SR2 next
                      MEMOUT next
                      MEMADD ← (MEMADD+1)<15:0>; OPRY<15:0> ← RTC<15:0> next
                      MEMOUT next
                      MEMADD ← (MEMADD+4)<15:0>; OPRY<15:0> ← RTC<31:16> next
                      MEMOUT next
                      MEMADD ← (MEMADD-2)<15:0> next
                      MEMREF next
                      SR1 ← MEMVAL<15:0>; MEMADD ← (MEMADD+1)<15:0> next
                      MEMREF next
                      SR2 ← MEMVAL<15:0>; MEMADD ← (MEMADD-2)<15:0> next
                      MEMREF next
                      (DECODE CLASS =>
                         (ITEMP ←ITEMP);         !to take care of CLASS=0
                         (ITEMP<3:0> ← I1; ITEMP<15:4> ← 0);
                         (ITEMP<3:0> ← I1; ITEMP<15:4> ← 0);
                         (ITEMP<6:0> ← I2<6:0>; ITEMP<15:7> ←0)
                      )next
                      P ← (MEMVAL+ITEMP)<15:0>
                     )
                 )
                )
         ) next
         INTFLG ← 0
        ));

!       I/O channel search operations
!       *****************************

   EXTINT :=             ! External interrupt
        (
     (DECODE PBITS[CTEMP] =>
        (CTEMP ← (CTEMP+1)<15:0> next
         EXTINT
        );
        (MEMADD ← (CTEMP+128)<15:0> next
         OPRY<15:0> ← INPUT[CTEMP] next
         MEMOUT; PBITS[CTEMP] ← 1; RBITS[CTEMP] ← 0
        )
     )
    ));

   DATA :=               ! Data word fetch or store
     (
      (DECODE QBITS[CTEMP] =>
          (CTEMP ← (CTEMP+1)<15:0> next DATA);
          (QBITS[CTEMP] ← 0 next IOPDT)
      )
     ));
```

```
!       UTILITIES / 6
!       -- -- - -----

!       I/O instruction fetch and execute
!       *********************************

    IOINST :=
      (
       (DECODE SBITS[CTEMP] =>
           (CTEMP ← (CTEMP+1)<15:0> next IOINST);
           (SBITS[CTEMP] ← 0 next
            (DECODE CTEMP<0> => CTEMP ← (CTEMP*G+G)<15:0>; CTEMP ← (CTEMP*G+2)<15:0>) next
            PCOUNT ← (IOCC[CTEMP]+1)<15:0>;
            MEMADD ← IOCC[CTEMP] next
            MEMREF next
            IR ← MEMVAL<15:0> next
            CDCODE next
            IOCC[CTEMP] ← PCOUNT
           )
       )
      );

!       UN-ASSIGNED OP CODES
!       ********************

        FAULT1 := ( INTFLG←1; CLASS←2; I1←0);

!       NOOP FOR CODES 70 - 77
!       **********************

        NOOP := (TEMP ← TEMP);
```

```
!       INSTRUCTION REPERTOIRE
!       -- ------------------

!       A) LOAD INSTRUCTIONS
!       ********************

        LOADD :=
        (DOUSEL next
                TEMPD1<31:0> ← (OPRY@OPRY1)<31:0>; CARRY ← 0; OVRFLW ← 0 next
                ASELDW ; CCD
        );

        PARLD :=
        (MEMREF next
                PAR[OPPA<5:0>] ← MEMVAL<15:0>
        );

        PARST :=
        (OPRY<15:0> ← PAR[OPPA<5:0>] next
                MEMOUT
        );

        PARCNT :=
        ((IF (ITEMP<5:0> NEQ OPRA<13:8>) =>
                        ITEMP<5:0> ← (ITEMP<5:0>+1)<5:0>;
                        MEMADD ← (MEMADD+1)<15:0>;
                        OPRA<5:0> ← (OPRA+1)<5:0> next
                        (DECODE UNO => PARLD; PARST) next
                        PARCNT
                )
        );

!-----------------------------------------------------------------------
!OPCODE = 01

        LOAD :=
        (FDCODE next
                OPRA ← OPRY; CARRY ← 0; OVRFLW ← 0 next
                ASELW ; CC
        );

! END OF OPCODE 01
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!OPCODE = 54

        LOADPG :=
        (DECODE (FCODE EQL 2) =>
                (ASELR; FDCODE next
                        PAR[OPRA<5:0>] ← OPRY<15:0> next
                        (IF (FCODE EQL 3) => UNO ← 0; ITEMP ← 0 next PARCNT)
                );
                FAULT1
        );

! END OF OPCODE 54
!-----------------------------------------------------------------------
```

```
!       A) LOAD INSTRUCTIONS (CONT'D)
!       ****************************

! ----------------------------------------------------------------------
!OPCODE = 02

        COMP :=
        ((IF (FCODE EQL 0) =>
                        (DECODE MPEG =>
                                !Make positive
                PR :=           (ASELR next
                                        (IF OPRA<15> =>
                                                OPRA ← (MINUS OPRA)<16:0> next
                                                (IF (OPRA<15:0> EQL #100000) => OVRFLW ← 1; CARRY ← 1));
                                                (IF (OPRA<15:0> NEQ #100000) => OVRFLW ← 0; CARRY ← 0));
                                                ASELW ; CC
                                        )
                                ));
                                !Make negative
                NR :=           (ASELR next
                                        (IF (OPRA<15> EQL 0) AND (OPRA<14:0> NEQ 0) =>
                                                OPRA ← (MINUS OPRA)<15:0> next
                                                (IF OPRA<15:0> EQL #100001 => CARRY ← 1));
                                                (IF OPRA<15:0> NEQ #100001 => CARRY ← 0));
                                                OVRFLW ← 0; ASELW ; CC
                                        )
                                ));
                                !Round
                PREG :=         (ASELDR next
                                        (DECODE TEMPD1<31> =>
                                                (IF TEMPD1<15> =>
                                                        TEMPD1<32:16> ← (TEMPD1<31:16>+1)<16:0> next
                                                        (IF (TEMPD1<31:16> EQL #100000) => OVRFLW ← 1));
                                                        (IF (TEMPD1<31:16> NEQ #100000) => OVRFLW ← 0)
                                                ));
                                                (IF (NOT TEMPD1<15>) =>
                                                        TEMPD1<32:16> ← (TEMPD1<31:16>-1)<16:0> next
                                                        (IF (TEMPD1<31:16> EQL #77777) => OVRFLW ← 1));
                                                        (IF (TEMPD1<31:16> NEQ #77777) => OVRFLW ← 0)
                                                )
                                        ) next
                                        CARRY ← TEMPD1<32>;
                                        OPRA<16:0> ← TEMPD1<32:16> next
                                        ASELW ; CC
                                ));
                                FAULT));
```

```
!       A) LOAD INSTRUCTIONS (CONT'D)
!       ****************************

                        !Twos complement
        TCR :=          (ASELR next
                                OPRA ← (MINUS OPRA)<16:0> next
                                (IF (OPRA<15:0> EQL #100000) => OVRFLW ← 1);
                                (IF (OPRA<15:0> NEQ #100000) => OVRFLW ← 0);
                                CARRY ← (NOT OPRA<16>); ASELW ; CC
                        );
                        !Twos complement double
        TCDR :=         (ASELDR next
                                TEMPD1 ← (MINUS TEMPD1)<32:0> next
                                (IF (TEMPD1<31:0> EQL #20000000000) => OVRFLW ← 1);
                                (IF (TEMPD1<31:0> NEQ #20000000000) => OVRFLW ← 0);
                                CARRY ← (NOT TEMPD1<32>); ASELDW ; CCD
                        );
                        !Ones complement
        OCR :=          (ASELR next
                                OPRA<16:0> ← (NOT OPRA) next
                                CARRY ← 0; OVRFLW ← 0; ASELW ; CC
                        );
                        FAULT1;
                        !Increment Ra
        IRDR :=         (ASELR next
                                OPRA ← (OPRA+1)<16:0> next
                                CARRY ← OPRA<16>;
                                (IF (OPRA<15:0> EQL #100000) => OVRFLW ← 1);
                                (IF (OPRA<15:0> NEQ #100000) => OVRFLW ← 0);
                                ASELW ; CC
                        );
                        !Decrement Ra
        DRDR :=         (ASELR next
                                OPRA<16> ← 1 next
                                OPRA ← (OPRA-1)<16:0> next
                                CARRY ← (NOT OPRA<16>);
                                (DECODE (OPRA<14:0> EQL #77777) => OVRFLW ← 0; OVRFLW ← 1);
                                ASELW ; CC
                        );
                        !Increment Ra by two
        IRTR :=         (ASELR next
                                OPRA ← (OPRA+2)<16:0> next
                                (IF (OPRA<15:0> EQL #100000) OR (OPRA<15:0> EQL #100001) => OVRFLW ← 1);
                                (IF (OPRA<15:0> NEQ #100000) AND (OPRA<15:0> NEQ #100001) => OVRFLW ← 0);
                                CARRY ← OPRA<16>; ASELW ; CC
                        );
                        !Decrement Ra by two
        DRTR :=         (ASELR next
                                OPRA<16> ← 1 next
                                OPRA ← (OPRA-2)<16:0> next
                                CARRY ← (NOT OPRA<16>); ASELW ; CC;
                                (DECODE (OPRA<14:0> EQL #77777) OR (OPRA<14:0> EQL #77776) =>
                                        OVRFLW ← 0;
                                        OVRFLW ← 1
                                )
                        );
                        FAULT1;
                        FAULT1;
                        FAULT1;
                        FAULT1
                   )
        );
        (IF (FCODE EQL 1) OR (FCODE EQL 3) => LOAD0);
        (IF (FCODE EQL 2) => FAULT1)
    );

! END OF OPCODE 02
!---------------------------------------------------------------------
```

```
!       D) LOAD INSTRUCTIONS (CONT'D)
!       *****************************

!-  - ------ - -- -       ----  ---------------- ------- -----
!OPCODE = 03

UCTRL :=                          ! Unary Control and Load Multiple
( DECODE FCODE =>
        ( DECODE MREG =>
                                  ! Executive return
        ER :=   (INTFLG←1;CLASS←2;I1←6; OPPA<15:0> ← PCOUNT next
                 CARRY ← 0; OVRFLW ← 0;  ASELW; CC
                );

                                  ! Store SR1
        SSOR := (OPPA<15:0> ← SR1 next
                 CARRY ← 0; OVRFLW ← 0; ASELW; CC
                );

                                  ! Store SR2
        SSTR := (OPPA<15:0> ← SR2 next
                 CARRY ← 0; OVRFLW ← 0; ASELW; CC
                );

                                  ! Store RTC lower
        SCR :=  (OPPA<15:0> ← RTC<15:0> next
                 CARRY ← 0; OVRFLW ← 0; ASELW ; CC
                );

        LPR :=  ( ASELR next PCOUNT ← OPPA<15:0> );      ! Load P register
        LSOR := ( ASELR next SR1 ← OPPA<15:0> );         ! Load SR1
        LSTR := ( ASELR next SR2 ← OPPA<15:0> );         ! Load SR2
        LCR :=  ( ASELR next RTC ← OPPA<15:0> );         ! Load lower half of RTC
                                  ! Enable RTC
        ECR :=  ((IF (AREG EQL 0) => RTCE ← 1);
                 (IF (AREG NEQ 0) => FAULT))
                );

                                  ! Disable RTC
        DCR :=  ((IF (AREG EQL 0) => RTCE ← 0);
                 (IF (AREG NEQ 0) => FAULT))
                );

                                  ! Load and enable the Monitor Clock
        LEM :=  (ASELR next
                 MON ← OPPA<15:0> next
                 MONE ←1; MONINE ← 1
                );
```

```
!        A) LOAD INSTRUCTIONS (CONT'D)
!        ****************************

                                ! Disable Monitor Clock
        DMCP := (MONE ← 0; MONINE ← 0);

                                ! Load and enable Clock Double
        LCD :=  (ASELDP next
                 RTC ← TEMPD1<31:0> next
                 RTCE ← 1
                );

                                ! Store Clock Double
        SCD :=  (TEMPD1<31:0> ← RTC next
                 ASELDW; CCD
                );

                                ! Enable Clock Interrupt
        ECIR := ((IF (AREG EQL 0) => RTCOI ← 1);
                 (IF (AREG NEQ 0) => FAULTI)
                );

                                ! Disable Clock Interrupt
        DCIR := ((IF (AREG EQL 0) => RTCOI ← 0);
                 (IF (AREG NEQ 0) => FAULTI)
                )

        );
        FAULTI;
        FAULTI;
        ( MEMADD ← PCOUNT next           ! Load Multiple
          PCOUNT ← (PCOUNT+1)<15:0> next
          MEMREF next
          MEMADD ← MEMVAL<15:0> next
                ( DECODE (AREG EQL MREG) =>
        LOOP:=          ( MEMREF next
                          OPRA ← MEMVAL next
                          ASELW next
                          AREG ← (AREG+1)<3:0> ; MEMADD ← (MEMADD+1)<15:0>  next
                          (IF (AREG NEQ (MREG+1)<3:0>) => LOOP)
                        );
                        ( MEMREF next OPRA ← MEMVAL next ASELW )
                )
        )
);

! END OF OPCODE 03
!------------------------------------------------------------------
```

```
!       A) LOAD INSTRUCTIONS (CONT'D)
!       *****************************

!                    .  .    ..  ..........  ..  .  ........  .....
!OPCODE = 05

        LOADI :=
        !Set bit
        ((IF (FCODE EQL 0) =>
                        ASELP next
                        BITS[MREG] ← 1; OVRFLW ← 0; CARRY ← 0 next
                        ASELW ; CC
                );
                !Load and index
                (IF (FCODE EQL 1) OR (FCODE EQL 3) =>
                        LOAD next
                        (IF (AREG NEQ MREG) =>
                                (DECODE RSELCT => R0[MREG] ← (OPRM+1)<15:0>; R1[MREG] ← (OPRM+1)<15:0>)
                        )
                );
                (IF (FCODE EQL 2) => FAULT))
        );

! END OF OPCODE 05
!------ ------------------------------------------------------------------

!-------------------------------------------------------------------------
!OPCODE = 06

        ZEROBT :=
        !Zero bit
        ((IF (FCODE EQL 0) =>
                        ASELR next
                        BITS[MREG] ← 0; CARRY ← 0; OVRFLW ← 0 next
                        ASELW ;CC
                );
                !Load double and index by two
                (IF (FCODE EQL 1) OR (FCODE EQL 3) =>
                        LOADD next
                        (IF (MREG NEQ AREG) =>
                                (DECODE RSELCT => R0[MREG] ← (OPRM+2)<15:0>; R1[MREG] ← (OPRM+2)<15:0>)
                        )
                );
                (IF (FCODE EQL 2) => FAULT))
        );

! END OF OPCODE 06
!-------------------------------------------------------------------------

!-------------------------------------------------------------------------
!OPCODE = 0D
        BYTELD :=
        !Diagnostic return
        (DECODE FCODE =>
                (DECODE DJ =>
                        FAULT;
                        (DECODE RSELCT => MICROP ← R0[#17]; MICROP ← R1[#17])
                );
                FAULT;
                FAULT;
                !Byte load
                (BYTE ← 1 next RX1  next BYTRD next
                        OPRA ← OPRY; CARRY ← 0; OVRFLW ← 0 next
                        BYTE ← 0 ; ASELW ; CC
                )
        );

! END OF OPCODE 0A
!-------------------------------------------------------------------------
```

```
!         A) LOAD INSTRUCTIONS (CONT'D)
!         *****************************

!.
!OPCODE = 04

          BYTLDX :=
          (DECODE FCODE =>
                  ! Unary shift operations

                  ((IF (MREG EQL 1) =>              ! Reverse register
                          (TMP1 + 0; TMP2 + 15; ASELR next
                  REPEAT:= (TMPBITS[TMP1] + BITS[TMP2] next
                            TMP1+(TMP1+1)<1:0>; TMP2+(TMP2-1)<1:0> next
                            (IF (TMP2 GEQ 0) => REPEAT)
                           ) next
                           OPRA<15:0> + TEMP next
                           ASELW; CC; CARRY + 0; OVRFLW + 0
                          )
                   );
                   (IF (MREG EQL 2) =>
                          (ASELR ; TMP1 + 15; TEMP + 0 next
                  AGAIN:=  (TEMP + (TEMP+BITS[TMP1])<15:0> next
                            TMP1 + (TMP1-1)<1:0> next
                            (IF (TMP1 GEQ 0) => AGAIN)
                           ) next
                           OPRA<15:0> + TEMP; AREG + (AREG+1)<3:0> next
                           ASELW
                          )
                   );
                   (IF (MREG EQL 3) =>
                          (ASELOR next
                           AREG<3:0>+(AREG+2)<3:0> next
                           ASELR next
                           (IF (TEMPD1<31> NEQ TEMPD1<30>) =>
                  MORE:=            (TEMPD1<31:0>+(TEMPD1<31:0> ISL0 1)<31:0>;
                                     OPRA<15:0> + (OPRA+1)<15:0> next
                                     (IF (TEMPD1<31> NEQ TEMPD1<30>) => MORE)
                                    )
                           ) next
                           ASELW next
                           AREG<3:0>+(AREG - 2)<3:0> next
                           ASELOW
                          )
                   );
                   (IF ((MREG NEQ 1) AND (MREG NEQ 2) AND (MREG NEQ 3)) => FAULT1)
                  );
                  FAULT1;
                  FAULT1;

                  ! Byte load and index by 1

                  (BYTELD next
                   (IF (AREG NEQ MREG) =>
                          (DECODE RSELECT => R0[MREG] + (OPRM+1)<15:0>;
                                             R1[MREG] + (OPRM+1)<15:0>
                          )
                   )
                  )
          );

! END OF OPCODE 04
!----------------------------------------------------------------------------
```

```
[illegible]                                                              Page [illegible]

!        [illegible]
!        * ***************************.

!
[illegible] = 07

        (ICODE :=
        !Common exit
        ((IF (FCODE EQL 0) =>
                        ASELP next
                        SP1(8) ← BITS(MREG);
                        (IF MREG EQL #17 => SP1<9>←SP1<8>);
                        CARRY ← 0; OVFLW ← 0
                );
                !Load new
                (IF (FCODE EQL 1) OR (FCODE EQL 3) =>
                        FOCODE next
                        PCOUNT ← OPRY<15:0>; MEMADD ← (MEMADD+1)<15:0> next
                        MEMREF next
                        UNO ← MSELCT next          ! save old mselct
                        SP1 ← MEMVAL<15:0>; MEMADD ← (MEMADD+1)<15:0> next
                        DOS ← MSELCT next          ! save new mselct
                        MSELCT ← UNO next          ! restore old mselct
                        MEMREF next
                        SP2 ← MEMVAL<15:0> next
                        MSELCT ← DOS               ! restore new mselct
                );
                (IF (FCODE EQL 2) => FAULT))
        ))

! END OF OPCODE 07
!-------------------------------------------------------------------
```

```
!        B: STORE ... AND SHIFT INSTRUCTIONS
!        ********************************

         [illegible] := [illegible] next RI [illegible] next [illegible];

         STSIM :=
         (ASELR;
                  (IF (FCODE EQL 1) => RI), (IF (FCODE EQL 3) => RX) next
                  OPPY ← OPPA next
                  MEMOUT
         );

         S[illegible] :=
         ([illegible]R;
                  (IF (FCODE EQL 1) => RI); (IF (FCODE EQL 3) => RX) next
                  OPPY ← TEMPDI<32:16>; OPPYI ← TEMPDI<15:0> next
                  MEMOID
         );

         STOMUL :=
         (IF (APEG NEQ MPEG) =>
                  APEG ← (APEG+1)<3:0>; MEMADD ← (MEMADD-1)<15:0> next
                  ASELR next
                  OPPY ← OPPA next
                  MEMOUT next
                  STOM[illegible]
         );

!---------------------------------------------------------------------
!OPCODE = 10

         LGRSFT :=
         !Logical right shift
         ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                            ASELR; FDCODE next
                            OPPA ← (OPPA †SR0 OPPY<5:0>)<16:0>; CARRY ← 0; OVRFLW ← 0 next
                            ASELW ; CC
                  );
                  (IF (FCODE EQL 3) => BYTEST);
                  (IF (FCODE EQL 1) => FAULT)
         );

! END OF OPCODE 10
!---------------------------------------------------------------------

!---------------------------------------------------------------------
!OPCODE = 11

         ALPSFT :=
         !Algebraic right shift
         ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                            ASELR; FDCODE next
                            OPPA<16> ← OPPA<15> next
                            (DECODE OPPA<15> => OPPA ← (OPPA †SR0 OPPY<5:0>); OPPA ← (OPPA †SR1 OPPY<5:0>)) next
                            CARRY ← 0; OVRFLW ← 0; ASELW ; CC
                  );
                  (IF (FCODE EQL 1) OR (FCODE EQL 3) => STSING)
         );

! END OF OPCODE 11
!---------------------------------------------------------------------
```

```
!       B) STORE . AND SHIFT INSTRUCTIONS (CONT'D)
!       ****************************************


!- 
!OPCODE = 12

        DLGRST :=
        !Logical right double shift
        ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                        ASELOP; FDCODE next
                        TEMPD1 = (TEMPD1 ISR0 OPRY<5:0>); CARRY = 0; OVRFLW = 0 next
                        ASELOW ; CCD
                ));
                (IF (FCODE EQL 1) OR (FCODE EQL 3) => STDOUB)
        ));

! END OF OPCODE 12
!--------------------------------------------------------------------

!--------------------------------------------------------------------
!OPCODE = 13

        DALRST :=
        !Algebraic right double shift
        ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                        FDCODE; ASELOP next
                        TEMPD1<32> = TEMPD1<31> next
                        (DECODE TEMPD1<32> =>
                                TEMPD1 = (TEMPD1 ISR0 OPRY<5:0>);
                                TEMPD1 = (TEMPD1 ISR1 OPRY<5:0>)
                        ) next
                        CARRY = 0; OVRFLW = 0; ASELOW ; CCD
                ));
                (IF (FCODE EQL 1) => FAULT1);
                !Store multiple
                (IF (FCODE EQL 3) =>
                        ASELP; MEMADD = PCOUNT next
                        PCOUNT = (PCOUNT+1)<15:0> next
                        MEMREF next
                        MEMADD = MEMVAL<15:0>; OPRY = OPRA next
                        MEMOUT next
                        STOMUL
                )
        ));

! END OF OPCODE 13
!--------------------------------------------------------------------

!--------------------------------------------------------------------
!OPCODE = 14

        ALLSFT :=
        !Algebraic left shift
        ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                        ASELR; FDCODE next
                        UNO = OPRA<15> next
                        OPRA = (OPRA ISL0 OPRY<5:0>) next
                        CARRY = 0; OVRFLW = (UNO XOR OPRA<15>);  ASELW ; CC
                ));
                !Byte store  and index
                (IF (FCODE EQL 3) =>
                        BYTEST next
                        (IF (RREG NEQ MREG) =>
                                (DECODE RSELECT => R0[MREG] = (OPRM+1)<15:0>; R1[MREG] = (OPRM+1)<15:0>)
                        )
                )
        ));

! END OF OPCODE 14
!--------------------------------------------------------------------
```

```
!       B) STORE AND SHIFT INSTRUCTIONS (CONT'D)
!       ***************************************


!-----------------------------------------------------------------
!OPCODE = 15

        CPLSFT :=
        !Circular left shift
        ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                        ASELR; FDCODE next
                        OPPA<15:0> ← (OPPA<15:0> †RL OPRY<5:0>)<15:0> next
                        CARRY ← 0; OVRFLW ← 0; ASELW ; CC
                );
                !Store and index
                (IF (FCODE EQL 1) OR (FCODE EQL 3) =>
                        STSING next
                        (IF (AREG NEQ MREG) =>
                                (DECODE RSELCT => R0[MREG] ← (OPRM+1)<15:0>; R1[MREG] ← (OPRM+1)<15:0>)
                        )
                )
        );

! END OF OPCODE 15
!-----------------------------------------------------------------

!-----------------------------------------------------------------
!OPCODE = 16

        DALLST :=
        !Double left algbraic shift
        ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                        ASELDR; FDCODE next
                        UNO ← TEMPD1<31> next
                        TEMPD1<31:0> ← (TEMPD1 †SL0 OPRY<5:0>)<31:0>;
                        CARRY ← 0; OVRFLW ← (UNO XOR TEMPD1<31>); ASELDW ; CCD
                );
                !Store double and index by two
                (IF (FCODE EQL 1) OR (FCODE EQL 3) =>
                        STDOUB next
                        (IF (AREG NEQ MREG) =>
                                (DECODE RSELCT => R0[MREG] ← (OPRM+2)<15:0>; R1[MREG] ← (OPRM+2)<15:0>)
                        )
                )
        );

! END OF OPCODE 16
!-----------------------------------------------------------------

!-----------------------------------------------------------------
!OPCODE = 17

        DCPLST :=
        !Circular left double shift
        ((IF (FCODE EQL 0) OR (FCODE EQL 2) =>
                        ASELDR; FDCODE next
                        TEMPD1<31:0> ← (TEMPD1<31:0> †RL OPRY<5:0>)<31:0> next
                        CARRY ← 0; OVRFLW ← 0; ASELDW ; CCD
                );
                !Store zeros
                (IF (FCODE EQL 1) OR (FCODE EQL 3) =>
                        OPRY ← 0; (IF (FCODE EQL 1) => R1); (IF (FCODE EQL 3) => RX) next
                        MEMOUT
                )
        );

! END OF OPCODE 17
!-----------------------------------------------------------------
```

```
!       B) STORE. AND SHIFT INSTRUCTIONS (CONT'D)
!       ****************************************

!-  . . . .     . ... .. ... ....    ----------------------
!OPCODE = 55

        STADPG :=
        (DECODE FCODE =>
                (ASELR) PP next
                        (DECODE PSELCT => R0[MREG] ← PAR;OPRA<5:0>]; R1[MREG] ← PAR[OPRA<5:0>]]
                );
                (ASELR) RI next
                        PARST
                );
                FAULT];
                (ASELR) PX next
                        UNO ← 1; ITEMP ← 0 next
                        PARST next
                        PARCNT
                )
        );

! END OF OPCODE 55
!-----------------------------------------------------------------------
```

```
!       C) ARITHMETIC INSTRUCTIONS
!       **************************

!---------------------------------------------------------------------
!OPCODE = 20

        SUBTCT :=
        (ASELR; FDCODE next
                UNO ← OPPA<15> next
                OPPA ← (OPPA<15:0> MINUS OPRY<15:0>)<16:0> next
                CAPRY ← OPRA<16>;
                OVRFLW ← ((OPRY<15> XOR UNO) AND (OPPA<15> XOR UNO));
                ASELW ; CC
        );

! END OF OPCODE 20
!----------------------------------------------------------------------

!---------------------------------------------------------------------
!OPCODE = 22

        ADD :=
        (ASELR; FDCODE next
                UNO ← OPPA<15> next
                OPRA ← (OPPA<15:0>+OPRY<15:0>)<16:0> next
                CAPRY ← OPRA<16>;
                OVRFLW ← (((NOT UNO) XOR OPRY<15>) AND (UNO XOR OPRA<15>));
                ASELW ; CC
        );

! END OF OPCODE 22
!----------------------------------------------------------------------

!----------------------------------------------------------------------
!OPCODE = 21

        DSUBTCT :=
        ((IF (FCODE NEQ 2) =>
                        ASELDR; DOBSEL next
                        UNO ← TEMPD1<31> next
                        TEMPD1 ← (TEMPD1<31:0> MINUS OPRY<15:0>@OPRY1)<32:0> next
                        CARRY ← TEMPD1<32>;
                        OVRFLW ← ((UNO XOR OPRY<15>) AND (UNO XOR TEMPD1<31>));
                        ASELDW ; CCD
                );
                (IF (FCODE EQL 2) => FAULT))
        );

! END OF OPCODE 21
!----------------------------------------------------------------------

!----------------------------------------------------------------------
!OPCODE = 23

        DADD :=
        ((IF (FCODE NEQ 2) =>
                        ASELDR; DOBSEL next
                        UNO ← TEMPD1<31> next
                        TEMPD1 ← (TEMPD1<31:0>+OPRY<15:0>@OPRY1)<32:0> next
                        CAPRY ← TEMPD1<32>;
                        OVRFLW ← (((NOT UNO) XOR OPRY<15>) AND (UNO XOR OPRA<15>));
                        ASELDW ; CCD
                );
                (IF (FCODE EQL 2) => FAULT))
        ));

! END OF OPCODE 23
!----------------------------------------------------------------------

!----------------------------------------------------------------------
!opcode = 56    multiply double

        DMUL:=
        ((IF FCODE NEQ 2 =>
                        ASELDR;DOBSEL;CCDES←0 NEXT
                        TEMPD2 ← (OPRY<15:0>@OPRY1)<31:0> NEXT
                        TEMPD3 ← (TEMPD1<31:0>*TEMPD2<31:0>)<63:0> NEXT
                        (IF TEMPD1<31> => TEMPD3<63:32>←(TEMPD3<63:32>-TEMPD2)<31:0>) NEXT
                        (IF TEMPD2<31> => TEMPD3<63:32>←(TEMPD3<63:32>-TEMPD1)<31:0>) NEXT
                        TEMPD1 ← TEMPD3<63:32> NEXT
                        ASELDW NEXT
                        APLG ← (APLG+2)<3:0>;
                        TEMPD1 ← TEMPD3<31:0> NEXT
                        ASELDW;
                        (DECODE TEMPD3<63> =>
                            (IF TEMPD3<62:0> NEQ 0 => CCDES ← 1);
```

```
                                  (CCDES ← 3)
                              )
                    );
                    (IF FCODE EQL 2 => FAULT))
        );
! END OF OPCODE 56
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!OPCODE = 57

        DDVD:=
        ((IF FCODE NEQ 2 =>
                                  ASELDP;DDBSEL;CCDES←0 NEXT
                                  TEMPD3<63:32>← TEMPD1<31:0> NEXT
                                  APEG ← (APEG+2)<3:0> NEXT
                                  ASELDP NEXT
                                  TEMPD3<31:0> ← TEMPD1<31:0>; UNO ← TEMPD3<63> NEXT
                                  (IF UNO => TEMPD3 ← (MINUS TEMPD3)<63:0>);
                                  TEMPD2 ← OPRY<15:0>@OPRY1 NEXT
                                  (IF TEMPD2<31> => DOS ← NOT UNO; TEMPD2 ← (MINUS TEMPD2)<31:0>) NEXT
                                  DWQUOT ← (TEMPD3<63:0>/TEMPD2<31:0>)<63:0> NEXT
                                  (IF DWQUOT<63:31> NEQ 0 => OVRFLW ← 1; BAILOUT DDVD);
                                  TEMPD1 ← (TEMPD3 - (TEMPD2<31:0>*DWQUOT<31:0>)<63:0>)<31:0>;
                                  APEG ← (APEG-2)<3:0> NEXT

                                  (IF UNO => TEMPD1 ←(MINUS TEMPD1)<31:0>);
                                  (IF DOS => DWQUOT ←(MINUS DWQUOT)<31:0>) NEXT
                                  ASELDW NEXT
                                  APEG ← (APEG+2)<3:0>; TEMPD1←DWQUOT<31:0> NEXT
                                  ASELDW; CCD
                    ));
                    (IF FCODE EQL 2 => FAULT))
        ));
!END OF OPCODE 57
!-----------------------------------------------------------------------------
```

```
!       C) ARITHMETIC INSTRUCTIONS (CONT'D)
!       **********************************

!-------------------------------------------------------------------
!OPCODES = 60,61    FLOATING POINT SUBTRACT AND ADD

        FSUB:=
        ((IF (FCODE NEQ 2) =>
                        ASELDR; DOBSEL next
                        UNO ← TEMPD1<31> next
                        TEMPD1 ← (TEMPD1<31:0> MINUS OPRY<15:0>@OPRY1)<32:0> next
                        CARRY ← TEMPD1<32>;
                        OVRFLW ← ((UNO XOR OPRY<15>) AND (UNO XOR TEMPD1<31>));
                        ASELDW ; CCD
                );
                (IF (FCODE EQL 2) => FAULT1)
        );

        FADD:=
        ((IF (FCODE NEQ 2) =>
                        ASFLDR; DOBSEL next
                        UNO ← TEMPD1<31> next
                        TEMPD1 ← (TEMPD1<31:0>+OPRY<15:0>@OPRY1)<32:0> next
                        CARRY ← TEMPD1<32>;
                        OVRFLW ← (((NOT UNO) XOR OPRY<15>) AND (UNO XOR OPRA<15>));
                        ASELDW ; CCD
                );
                (IF (FCODE EQL 2) => FAULT1)
        );

! END OF FLOATING-POINT SUBTRACT AND ADD
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 64

        BYTSUB :=
        ((IF (FCODE NEQ 3) => FAULT1);
                (IF (FCODE EQL 3) =>
                        BYTE ← 1 next RXI next BYTRD next
                        OPPA<16> ←1; UNO ← OPRA<15> next
                        OPRA ← (OPRA-OPRY)<16:0> next
                        CARRY ← (NOT OPRA<16>);
                        OVRFLW ← (UNO XOR OPRA<15>);
                        BYTE ← 0 ; ASELW ; CC
                )
        );

! END OF OPCODE 64
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 65


        BYTADD :=
        ((IF (FCODE NEQ 3) => FAULT1);
                (IF (FCODE EQL 3) =>
                        BYTE ← 1 next RXI next BYTRD next
                        UNO ← OPPA<15> next
                        OPRA ← (OPPA+OPRY)<16:0> next
                        CARRY ← OPRA<16>; OVRFLW ← (UNO XOR OPRA<15>); BYTE ← 0 ; ASELW ; CC
                )
        );

! END OF OPCODE 65
!-------------------------------------------------------------------
```

```
!       C) ARITHMETIC INSTRUCTIONS (CONT'D)
!       **********************************

        DIVMOD:=
        (TEMPD2<31:0> ← (TEMPD1<31:0>/OPRY<16:0>)<31:0> next
         (IF (TEMPD2<31:15> NEQ 0) => OVRFLW ← 1) next
         OPPA<15:0> ← TEMPD2<15:0> next
         TEMPD2<31:0> ← (OPPA<15:0> * OPRY<16:0>)<31:0> next
         TEMPD1<31:16> ← (TEMPD1<31:0> - TEMPD2<31:0>)<15:0> next
         (IF ABC =>
                (OPPA<15:0> ← (MINUS OPPA<15:0>)<15:0> next
                 TEMPD1<31:16> ← (MINUS TEMPD1<31:16>)<15:0>
                )
         ) next
         TEMPD1<15:0> ← OPPA<15:0>
        );

!----------------------------------------------------------------------
!OPCODE = 26

        MULT :=
        (FDCODE; ASELDP next
                UNO ← (OPRY<15> XOR TEMPD1<15>) next
                (IF OPRY<15> => OPRY ← (MINUS OPRY)<16:0>);
                (IF TEMPD1<15> => TEMPD1<15:0> ← (MINUS TEMPD1<15:0>)<15:0>) next
                TEMPD1<31:0> ← (OPRY<15:0>*TEMPD1<15:0>);
                CARRY ← 0; OVRFLW ← 0 next
                (IF UNO => TEMPD1<31:0> ← (MINUS TEMPD1)<31:0>) next
                ASELDW ; CCD
        );

! END OF OPCODE 26

!----------------------------------------------------------------------
!OPCODE = 27

        DIVIDE:=
        (FDCODE ; ASELDP next
         UNO ← OPRY<15> ; DOS ← TEMPD1<31>; CARRY ← 0 ; OVRFLW ← 0 next
         ABC ← (UNO XOR DOS) next
         (IF UNO =>
                (OPRY<15:0>←(MINUS OPRY<15:0>)<15:0> ; OPRY<16>←0)
         );
         (IF DOS =>
                (TEMPD1<31:0>←(MINUS TEMPD1<31:0>)<31:0>;TEMPD1<32>←0)
         ) next
         (DECODE (OPRY EQL 0) =>
                ((IF TEMPD1<31> => OVRFLW ← 1) next
                 DIVMOD next
                 ASELDW ; CCD
                );
                OVRFLW ← 1
         )
        );

! END OF OPCODE 27
!----------------------------------------------------------------------
```

```
!       C) ARITHMETIC INSTRUCTIONS (CONT'D)
!       ***********************************

!-------------------------------------------------------------------------
!OPCODE = 52

        FMUL:=
        ((IF (FCODE NEQ 2) =>
                ASELOP; DOBSEL next
                TEMPD2 ← (OPRY<16:0>@OPRY1<15:0>) next
                UNO ← (TEMPD1<31> XOR TEMPD2<31>) next
                (IF TEMPD1<31> => TEMPD1 ← (MINUS TEMPD1)<32:0>);
                (IF TEMPD2<31> => TEMPD2 ← (MINUS TEMPD2)<32:0>) next
                TEMPD3<63:0> ← (TEMPD1<31:0> * TEMPD2<31:0>) next
                TEMPD1<31:0> ← (TEMPD3<63:0> SR0 16)<31:0> next
                CARRY ← 0; OVRFLW ← 0 next
                (IF UNO => TEMPD1<31:0> ← (MINUS TEMPD1<31:0>)<31:0>) next
                ASELOW; CCD
         );
         (IF (FCODE EQL 2) => FAULT))
        );

!END OF OPCODE 52
!-------------------------------------------------------------------------

!-------------------------------------------------------------------------
!OPCODE = 53

        FDIV:=
        ((IF (FCODE NEQ 2) =>
                ASELOP; DOBSEL next
                TEMPD2 ← (OPRY<16:0>@OPRY1<15:0>) next
                DOS ← TEMPD1<31>; UNO ← TEMPD2<31>; CARRY ← 0; OVRFLW ← 0 next
                ABC ← (DOS XOR UNO) next
                (IF DOS => (TEMPD1<31:0>←(MINUS TEMPD1<31:0>)<31:0>;TEMPD1<32>←0));
                (IF UNO => (TEMPD2<31:0>←(MINUS TEMPD2<31:0>)<31:0>;TEMPD2<32>←0)) next
                (DECODE (TEMPD2 EQL 0) =>
                        ((IF TEMPD1<31> => OVRFLW ← 1) next
                         TEMP ← 0 next
                !multiply the dividend by 2**16
                         TEMPD3<47:0> ← (TEMPD1<31:0>@TEMP<15:0>)<47:0> next
                         TEMPD1<31:0> ← (TEMPD3<47:0>/TEMPD2<31:0>)<31:0> next
                         (IF ABC => TEMPD1<31:0> ← (MINUS TEMPD1)<31:0>) next
                         ASELOW ; CCD
                        );
                        OVRFLW ← 1
                )
         );
         (IF (FCODE EQL 2) => FAULT))
        );

!END OF OPCODE 53
!-------------------------------------------------------------------------
```

```
!       D) LOGICAL INSTRUCTIONS
!       ***********************

! -------------------------------------------------------------------
!OPCODE = 24

        COMPAR :=
        (FDCODE; ASELR next
                UNO ← OPRA<15> next
                OPPA ← (OPRA<15:0> MINUS OPRY<15:0>)<16:0> next
                CARRY ← OPRA<16>;
                OVRFLW ← ((UNO XOR OPRY<15>) AND (UNO XOR OPRA<15>)) next
                (IF ((NOT(OPRA<15> XOR OVRFLW)) AND (OPRA<14:0> EQL 0)) => CCDES ← 0);
                (IF (OPRA<15> XOR OVRFLW) => CCDES ← 3);
                (IF (NOT(OPRA<15> XOR OVRFLW)) AND (OPRA<14:0> NEQ 0) => CCDES ← 1)
        );

! END OF OPCODE 24
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 25

        DCOMPR :=
        ((IF (FCODE EQL 2) => FAULT1);
                (IF (FCODE NEQ 2) =>
                        ASELDR; DOBSEL next
                        UNO ← TEMPD1<31> next
                        TEMPD1 ← (TEMPD1<31:0> MINUS OPRY<15:0>@OPRY1)<32:0> next
                        CARRY ← TEMPD1<32>;
                        OVRFLW ← ((UNO XOR OPRY<15>) AND (UNO XOR TEMPD1<31>)) next
                        (IF ((NOT(TEMPD1<31> XOR OVRFLW)) AND (TEMPD1<30:0> EQL 0)) => CCDES ← 0);
                        (IF (TEMPD1<31> XOR OVRFLW) => CCDES ← 3);
                        (IF ((NOT(TEMPD1<31> XOR OVRFLW)) AND (TEMPD1<30:0> NEQ 0)) => CCDES ← 1)
                )
        );

! END OF OPCODE 25
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 33

        MBKBUM :=
        (FDCODE; ASELDR next
                OPRA<15:0> ← ((TEMPD1<15:0> XOR TEMPD1<31:0>) OR (TEMPD1<15:0> AND OPRY<15:0>))<15:0> next
                CARRY ← 0; OVRFLW ← 0; ASELW ; CC
        );

! END OF OPCODE 33
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 34

        COMMSK :=
        (ASELDR; FDCODE next
                OPRA<15:0> ← (TEMPD1<15:0> AND TEMPD1<31:16>);
                OPRY<15:0> ← (TEMPD1<15:0> AND OPRY<15:0>);
                CARRY ← 0; OVRFLW ← 0 next
                (IF (OPRA EQL OPRY) => CCDES ← 0);
                (IF (OPRA GTR OPRY) => CCDES ← 1);
                (IF (OPRA LSS OPRY) => CCDES ← 3)
        );

! END OF OPCODE 34
!-------------------------------------------------------------------
```

```
!       D) LOGICAL INSTRUCTIONS (CONT'D)
!       ******************************

!-------------------------------------------------------------------
!OPCODE = 30

        PAND :=
        (ASELP; FOCODE next
                OPPA ← (OPPA AND OPRY); CAPRY ←0 ; OVRFLW ← 0 next
                ASELW ; CC
        );

! END OF OPCODE 30
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 31

        POR :=
        (ASELP; FOCODE next
                OPPA ← (OPRA OR OPRY); CAPRY ← 0; OVRFLW ← 0 next
                ASELW ; CC
        );

! END OF OPCODE 31
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 32

        PXOR :=
        (ASELR; FOCODE next
                OPRA ← (OPRA XOR OPRY); CARRY ← 0; OVRFLW ← 0 next
                ASELW ; CC
        );

! END OF OPCODE 32
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 66

        BYTCOM :=
        ((IF (FCCDE NEQ 3) => FAULT));
                (IF (FCODE EQL 3) =>
                        ASELP ; BYTE ← 1 next
                        PX1 next BYTRD next
                        BYTE ← 0 ;
                        (DECODE OPRA<15> =>
                                (DECODE (OPRA TST OPRY) => CCDES ← 3; CCDES ← 0; CCDES ← 1);
                                CCDES ← 9
                        )
                )
        );

! END OF OPCODE 66
!-------------------------------------------------------------------

!-------------------------------------------------------------------
!OPCODE = 67

        USERMC :=
        !User macros
        ((IF (FCODE NEQ 3) => FAULT));
                !Byte compare and index
                (IF (FCODE EQL 3) =>
                        BYTCOM next
                        (DECODE PSELCT => P0(MREG) ← (OPRM+1)<15:0>; P1(MREG) ← (OPRM+1)<15:0>)
                )
        );

! END OF OPCODE 67
!-------------------------------------------------------------------
```

```
!       E) JUMP INSTRUCTIONS
!       ********************

        LCJUMP :=
        ((DECODE SCODE => OPRY<15:7> + 0; OPRY<15:7> + #777) ; OPRY<6:0> + DCODE  next
                PCOUNT + (PCOUNT + OPRY<15:0>)<15:0>
        );

!------------------------------------------------------------------
!OPCODE = 40
        JUMP :=
        (DECODE (FCODE EQL 1) =>
                (FDCODE next
                        (DECODE AREG =>
                JEQL :=         (IF (CCDES EQL 0) => PCOUNT + OPRY<15:0>);
                JNEQ :=         (IF (CCDES NEQ 0) => PCOUNT + OPRY<15:0>);
                JGEQ :=         (IF (CCDES EQL 1) OR (CCDES EQL 0) => PCOUNT + OPRY<15:0>);
                JLSS :=         (IF (CCDES EQL 3) => PCOUNT + OPRY<15:0>);
                JOVP :=         (IF OVRFLW => PCOUNT + OPRY<15:0>);
                JCAR :=         (IF CARRY => PCOUNT + OPRY<15:0>);
                JPOT :=         (IF POT => PCOUNT + OPRY<15:0>);
                JBST :=         (IF BST => PCOUNT + OPRY<15:0>);
                JMP :=          (PCOUNT + OPRY<15:0>);
                JSTP :=         (STP + 1; PCOUNT + OPRY<15:0>);
                JSTP1 :=        (IF STOP1 => STP + 1; PCOUNT + OPRY<15:0>);
                JSTP2 :=        (IF STOP2 => STP + 1; PCOUNT + OPRY<15:0>);
                                (PCOUNT + OPRY<15:0>);
                                (PCOUNT + OPRY<15:0>);
                                (PCOUNT + OPRY<15:0>);
                                (PCOUNT + OPRY<15:0>)
                        )
                );
                LCJUMP
        );
! END OF OPCODE 40
!------------------------------------------------------------------

!------------------------------------------------------------------
!OPCODE = 42
        JMPLNK :=
        (DECODE (FCODE EQL 1) =>
                (FDCODE; OPRA + PCOUNT<15:0> next
                        (DECODE RSELCT => R0[AREG] + OPRA<15:0>; R1[AREG] + OPRA<15:0>) next
                        PCOUNT + OPRY<15:0>
                );
                FAULT1
        );
! END OF OPCODE 42
!------------------------------------------------------------------

!------------------------------------------------------------------
!OPCODE = 41
        JUMPIX :=
        !Index Jump
        (DECODE (FCODE EQL 1) =>
                (ASCLR next FDCODE next
                        (IF (OPRA NEQ 0) =>
                                OPRA + (OPRA-1)<15:0> next
                                PCOUNT + OPRY<15:0>;
                                (DECODE PSELCT => R0[AREG] + OPRA<15:0>; R1[AREG] + OPRA<15:0>)
                        )
                );
                !Indirect local Jump
                ((DECODE SCODE => OPRY<15:7> + 0; OPRY<15:7> + #777) ; OPRY<6:0> + DCODE  next
                        MEMADD + (PCOUNT+OPRY<15:0>)<15:0> next
                        MEMREF next
                        PCOUNT + MEMVAL<15:0>
                )
        );
! END OF OPCODE 41
!------------------------------------------------------------------
```

```
!       E) JUMP INSTRUCTIONS (CONT'D)
!       ******************************

!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!OPCODE = 43

        JMPLEM :=
        ((IF (FCODE EQL 0) => FAULT));
                (IF (FCODE EQL 1) =>
                        !Local
                        (decode scode => opry<15:7>+0; opry<15:7>+#777);
                        opry<7:0>+dcode next
                        MEMADD + (PCOUNT+OPRY<15:0>)<15:0> next
                        OPRY<15:0> + PCOUNT next
                        MEMOUT;PCOUNT + (MEMADD+1)<15:0>
                );
                (IF (FCODE EQL 2) OR (FCODE EQL 3) =>
                        FDCODE next
                        MEMADD + OPRY<15:0> next
                        OPRY<15:0> + PCOUNT next
                        MEMOUT;
                        PCOUNT + (MEMADD+1)<15:0>
                )
        );

! END OF OPCODE 43
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!OPCODE = 44

        JMPZRO :=
        (DECODE (FCODE EQL 1) =>
                (ASCLR; FDCODE next
                        (IF (OPRA EQL 0) => PCOUNT + OPRY<15:0>)
                );
                !Local jump equ!
                (IF (CCODES<0> EQL 0) => LCJUMP)
        );

! END OF OPCODE 44
!------------------   ---------------------------------------------------
```

```
!       C) JUMP INSTRUCTIONS (CONT'D)
!       ****************************

!                                                                   -----
!OPCODE = 45

        JMPNZP :=
        (DECODE (FCODE EQL 1) =>
                (ASELR) FDCODE next
                        (IF (OPPA NEQ 0) => PCOUNT + OPRY<15:0>)
                );
                !Local jump not equal
                (IF (CCDES<0> NEQ 0) => LCJUMP)
        );

! END OF OPCODE 45
!------------------------------------------------------------------- ----

!-----------------------------------------------------------------------
!OPCODE 46

        JMPPOS :=
        (DECODE (FCODE EQL 1) =>
                (ASELR) FDCODE next
                        (IF (OPPA<15> EQL 0)  => PCOUNT + OPRY<15:0>)
                );
                (IF (CCDES EQL 1) OR (CCDES EQL 0) => LCJUMP)
        );

! END OF OPCODE 46
!------------------------------------------------------------------ -

!-------------------------------------------------- -------------------
!OPCODE = 47

        JMPNEG :=
        (DECODE (FCODE EQL 1) =>
                (ASELR) FDCODE next
                        (IF (OPPA<15> EQL 1) => PCOUNT + OPRY<15:0>)
                );
                !Local jump less then
                (IF (CCDES EQL 3) => LCJUMP)
        );

! END OF OPCODE 47
!-----------------------------------------------------------------------
```

```
[illegible]

        [illegible]
        ---------------- -------

!                                                         ---
(OPCODE = 60

        !Literal right shift
        LITRSE :=
         (CARRY ← 0; OVRFLW ← 0;
                (DECODE FCODE =>
                LLRS := (ASELR next
                                OPRA<15:0> ← (OPRA<15:0> SR0 MREG)<15:0>;
                                ASELW ; CC
                        );
                LARS := (ASELR next
                                (DECODE OPRA<15> =>
                                        OPRA<15:0> ← (OPRA<15:0> SR0 MREG)<15:0>;
                                        OPRA<15:0> ← (OPRA<15:0> SR1 MREG)<15:0>
                                ) next
                                ASELW ; CC
                        );
                LLRD := (ASELDR next
                                TEMPD1← (TEMPD1 SR0 MREG) next
                                ASELDW ; CCD
                        );
                LARD := (ASELDR next
                                (DECODE TEMPD1<31> =>
                                        TEMPD1<31:0> ← (TEMPD1<31:0> SR0 MREG)<31:0>;
                                        TEMPD1<31:0> ← (TEMPD1<31:0> SR1 MREG)<31:0>
                                ) next
                                ASELDW ; CCD
                        )
                )
        );

! END OF OPCODE 60
!-- ---  --- - -  -- -- ------------ ---------- ----------------------

!                                                       --------------
(OPCODE = 61

        !Literal left shift
        LITLSH :=
        (CARRY ← 0;
                (DECODE FCODE =>
                LALS := (ASELR next
                                UNO ← OPRA<15> next
                                OPRA ← (OPRA<15:0> SL0 MREG) next
                                OVRFLW ← (UNO XOR OPRA<15>); ASELW ; CC
                        );
                LCLS := (ASELR next
                                OPRA<15:0> ← (OPRA<15:0> RL MREG);
                                OVRFLW ← 0 next
                                ASELW ; CC
                        );
                LALD := (ASELDR next
                                UNO ← TEMPD1<31> next
                                TEMPD1 ← (TEMPD1 SL0 MREG) next
                                OVRFLW ← (UNO XOR TEMPD1<31>); ASELDW ; CCD
                        );
                LCLD := (ASELDR next
                                TEMPD1<31:0> ← (TEMPD1<31:0> RL MREG) next
                                OVRFLW ← 0; ASELDW ; CCD
                        )
                )
        );

! END OF OPCODE 61
!-  --   --     ------ - -  -------- --- -------- ---------- ------
```

```
        ! [illegible]
        ******************************

!OPCODE = 62
        LITSUB :=         !Literal add and subtract
        (OPRY<16:4> ← 0; OPRY<3:0> ← MREG; OPRY1<15:0> ← 0 next
                (DECODE FCODE =>
                LSU :=  (ASELR next
                                OPRA<16> ← 1 next
                                OPRA ← (OPRA OPRY)<16:0> next
                                CARRY ← (NOT OPRA<16>); OVRFLW ← (UNO XOR OPRA<15>); ASELW ; CC
                        );
                LSUD := (ASELDR next
                                TEMPD1<32> ← 1 next
                                TEMPD1 ← (TEMPD1 OPRY1@OPRY)<32:0> next
                                CARRY ← (NOT TEMPD1<32>); OVRFLW ← (UNO XOR TEMPD1<31>); ASELDW ; CCD
                        );
                LA :=   (ASELR next
                                UNO ← OPRA<15> next
                                OPRA ← (OPRA+OPRY)<16:0> next
                                CARRY ← OPRA<16>; OVRFLW ← (UNO XOR OPRA<15>); ASELW ; CC
                        );
                LAD :=  (ASELDR next
                                UNO ← TEMPD1<31> next
                                TEMPD1 ← (TEMPD1 + OPRY1@OPRY)<32:0> next
                                CARRY ← TEMPD1<32>; OVRFLW ← (UNO XOR TEMPD1<31>); ASELDW ; CCD
                        )
                )
        );
! END OF OPCODE 62
!------------------------------------------------------------------
!------------------------------------------------------------------
!OPCODE = 63
        LITLCM :=                             !Literal Operations
        (OPRY<16:4> ← 0; OPRY<3:0> ← MREG next
                (DECODE FCODE =>
                        !Literal load
                LL :=   (OPRA ← OPRY; CARRY ← 0; OVRFLW ← 0 next
                                ASELW ; CC
                        );
                        !Literal compare
                LC :=   (ASELR next
                                (DECODE OPRA<15> =>
                                        (DECODE (OPRA TST OPRY) => CCDES ← 3; CCDES ← 0; CCDES ← 1);
                                        (CCDES ← 3)
                                )
                        );
                ! literal multiply
                LMUL := (ASELDR next
                        TEMPD1<31:0> ← (OPRY<15:0> * TEMPD1<15:0>));
                        CARRY ← 0; OVRFLW ← 0 next
                        ASELDW; CCD
                        );
                ! literal divide
                LDIV := (ASELDR next
                        DOS ← TEMPD1<31> ; CARRY ← 0 ; OVRFLW ← 0 next
                        (IF DOS =>
                                (TEMPD1<31:0> ← (MINUS TEMPD1<31:0>)<31:0>;
                                 TEMPD1<32> ← 0
                                )
                        );
                        OPRY<3:0> ← MREG ; OPRY<16:4> ← 0 next
                        (DECODE (OPRY EQL 0) =>
                                ((IF TEMPD1<31> => OVRFLW ← 1) next
                                 DIVMOD next
                                 ASELDW ; CC
                                );
                                OVRFLW ← 1
                        )
                        )
                )
        );
! END OF OPCODE 63
!         ---------------------------------------------
```

```
!
!OPCODE 35

        IO COM :=
        (
                !I/O COMMAND
        IOCP := (IF  FCODE EQL 0) =>
                IOE ← 1; P ← PCOUNT next
                PCOUNT ← #140 next
                MEMADD ← #140 next
                MEMREF next
                IR ← MEMVAL(15:8); OPRY(13:0) ← MEMVAL(13:0); OPRY(15:14) ← 0 next
                MEMOUT next
        ! execute I/O instruction here
                PCOUNT ← P next
                bailout (error
                 );
                !Biased fetch
        BF :=   (IF (FCODE EQL 1) OR (FCODE EQL 3) =>
                        FOLODE next
                        SP1(9) ← OPRY(15);
                        (IF (OPRY(15) EQL 1) => SP1(8) ← 1) next
                        OPRY(15:14) ← 3 next
                        MEMOUT
                );
                !Remote execute
        REX :=  (IF (FCODE EQL 2) =>
                        PI next
                        MEMADD ← OPRY(15:0) ;
                        PCOUNT ← (OPRY(15:0)+1)(15:0) next
                        PTEMP ← PCOUNT next
                        MEMREF next
                        IR ← MEMVAL(15:0) next
                        REXFLG ← 1
                )
        );

! END OF OPCODE 35
!-------------------------------------------------------------------------
```

```
!       OPCODE DECODING
!       ---------------
        OPCODE :=
        (CASE OPCODE OF =>
!                                       OP      PAGE
                    BYTELD;             '100    16
                    LOAD;               '101    11
                    COMP;               '102    12-13
                    UCTRL;              '103    14-15
                    BYTLDX;             '104    17
                    LOADI;              '105    16
                    ZEROBT;             '106    16
                    LDXOP;              '107    18
                    LGPSET;             '110    19
                    ALPSET;             '111    19
                    DLGPST;             '112    20
                    DALPST;             '113    20
                    ALLSET;             '114    20
                    CPLSET;             '115    21
                    DALLST;             '116    21
                    DCPLST;             '117    21
                    SUBTCT;             '120    23
                    DSBTCT;             '121    23
                    ADD;                '122    23
                    DADD;               '123    23
                    COMPAR;             '124    27
                    DCOMPR;             '125    27
                    MULT;               '126    25
                    DIVIDE;             '127    25
                    PAND;               '130    28
                    POP;                '131    28
                    PXOR;               '132    28
                    MSFSUB;             '133    27
                    COMMSK;             '134    27
                    IOCCOM;             '135    34
                    FAULTI;             '136  Not assigned
                    FAULTI;             '137  Trig and Hyper functions
                    JUMP;               '140    29
                    JUMPIX;             '141    29
                    JMPLNE;             '142    29
                    JMPLKM;             '143    30
                    JMPZRO;             '144    31
                    JMPNZR;             '145    31
                    JMPPOS;             '146    31
                    JMPNEG;             '147    31
                    FSUB;               '150    24
                    FADD;               '151    24
                    FMUL;               '152    26
                    FDIV;               '153    26
                    LOADPG;             '154    11
                    STADPG;             '155    22
                    DMUL;               '156 Double multiply
                    DDVD;               '157 Double divide
                    LITPSF;             '160    32
                    LITLSF;             '161    32
                    LITSUB;             '162    33
                    LITCM;              '163    33
                    BYTSUB;             '164    24
                    BYTADD;             '165    24
                    BYTCOM;             '166    28
                    USERMC;             '167    28
                    FAULTI;             '170  Input/Output Instructions (OPCODES 70 to 77)
                    FAULTI;             '171
                    FAULTI;             '172
                    FAULTI;             '173
                    FAULTI;             '174
                    FAULTI;             '175
                    FAULTI;             '176
                    FAULTI              '177
            );

        CDCODE := (TEMP + TEMP);                ! NOOP for I/O Instructions
        BRERO.=: [illegible] ← TEMP;            !Dummy for breakpoint feature

        EXECUTED

!       MACHINE CYCLE
!       ------- -----

        CYCLE :=
        ( MEMADD ← P next
                PCOUNT ← (P+1)<15:0> next
                MEMREF next
                IP ← MEMVAL<15:0> next
        RPT:=(  PFXFLG ← 0 next
                OPDCOD next
                (DECODE PFXFLG =>
                        (IF ((PCOUNT EQL PTEMP) OR (PCOUNT EQL (PTEMP+1)<15:0>))=>
                                (PCOUNT ← (P+2)<15:0> ; PTEMP ← 0)
                        );
                        RPT
                )
              ) next
                P ← PCOUNT next
                (DECODE STP=>
                        (INTRPT next
                         (IF (BRKPNT EQL P) => BREAK) next
                         CYCLE
                        );
                        STP ← 0
                )
         )
)

! END OF UYK20.ISP
```

Appendix A:
Phase II Comparative Evaluation of MCF Computer Architectures

Phase II
Comparative Evaluation of the MCF Computer Architectures

S.H. Fuller, G. Mathew, and L. Szewerenko

January 15,1978
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

This work has been supported by the Army Research Office under contract number DAAG29-77-C-0034.

Abstract

This study was undertaken to determine the relative efficiency of the following architectures:

UYK-7
UYK-19
UYK-20
GYK-12
AYQ-21 (PDP-11)

It is part of the second phase of a study being conducted under the Army/Navy Military Computer Family (MCF) program to determine the relative life cycle cost of Army/Navy computer based systems as a function of computer architecture. In Phase I of this study, an Army/Navy Computer Family Architecture (CFA) committee recomended, in August 1976, that the PDP-11 architecture be adopted as a future standard military computer architecture. The other four architectures are the ones in most common use today in Army/Navy computer applications.

From a set of 160 test programs (16 different algorithms) written by 16 different programmers we found:

Program Size	*Execution Efficiency*	
	Memory Activity	Processor Activity
(S measure)	(M measure)	(R measure)
PDP-11 (0.82)	UYK-20 (0.73)	UYK-20 (0.77)
UYK-20 (0.89)		
UYK-19 (0.93)	PDP-11 (0.88)	GYK-12 (0.96)
	GYK-12 (0.96)	PDP-11 (1.03)
GYK-12 (1.14)		UYK-7 (1.12)
UYK-7 (1.30)	UYK 19 (1.18)	UYK-19 (1.17)
	UYK-7 (1.38)	

In each column the five architectures are ranked according to performance in that particular measure. The S measure is a measure of relative program size, M measures the relative number of memory accesses used and R measures the relative number of CPU operations needed. The architectures are clustered into groups based on gaps in performance which were statistically significant at a practical level (i.e, the gaps in performance were statistically significant at the 95% confidence level). The numbers in parenthesis give the average performance for an architecture in this study. For example, a machine with an S measure of 0.80 would require only 80% of the memory required by the average of these machines (20% less than average), while one with an S measure of 1.50 would require 50% more memory than the average.

Introduction

The question of the relative cost effectiveness of several different computers in the same application has traditionally been answered by the use of benchmark programs executed on the candidate machines. This technique unfortunately confounds the efficiency of the instruction set with the speed of the hardware used to implement it. Advances in hardware technology will, more often than not, obsolete the hardware long before the usefulness of the software declines. In such cases the question of long term cost effectiveness can only be answered in terms of the efficiency of the instruction set. An efficient instruction set will be amenable to cost effective implementation in state of the art technologies at any point of the software's life cycle.

The purpose of this study is to evaluate the efficiency of several computer architectures independently of their hardware implementations. The following definition of computer architecture was used in this study (and is the same definition as used by the CFA Committee [MC76]):

> Computer Architecture: The structure of the computer a programmer needs to know in order to write any time independent, machine language program that will run correctly on the computer.

Thus an efficient architecture will have the property that a hardware realization of the architecture will be more cost effective than a technologically similar realization of a less efficient architecture.

The results of this evaluation will be joined with the concurrently proceeding Software Support Evaluation and Life Cycle Cost Evaluation.Together, they will provide an analysis of the cost effectiveness of selecting each of the MCF architectures (UYK20, GYK12, UYK19, UYK7, PDP11) for implementation as a family of machines for use in Army and Navy Applications.

Overview

The methodology used in this study is based on a similar previous study for the CFA Committee comparing alternative commercial architectures [FU76]. However, several significant improvements have been made in the methodology of this second study. Briefly, the differences are:

1. The set of test programs has been improved to be more uniform in size and wider in scope; The individual tests are more precisely directed at architectural features.

2. The dynamic program measures have been extended to provide information on implementability over a range of hardware parallelism, as well as hardware speed.

3. The processor activity measure has been completely redefined. The

original R measure was found to be highly correlated with the original memory activity measure, and thus provided little additional information. It also failed to capture the inherent cost differences between simple and complex processor computations.

4. The method of computing program measures has been automated.

5. A superior statistical design was chosen which allowed more significant results to be extracted from the program measures.

A set of test programs was selected to test significant applications or capabilities of the architectures. Each program was described in a Program Description Language (PDL) which specified the algorithm to be used but left unspecified the exact machine level implementation of the algorithm. All test programs were designed to be writable by a test programmer in one or two pages of machine code.

Sixteen test programmers were selected to write test programs for the five MCF Architectures. Each programmer was assigned two programs to be implemented on all five architectures. The assignment was done according to a statistical design which attempted to separate architecture effects from programmer and program effects. The programs coded by the test programmers were executed using a standard set of test data on an ISP simulator written for each machine. The ISP simulator gathered statistics on the execution of the programs. Measures of efficiency computed from these statistics were used in an analysis of variance to determine the relative efficiency of each architecture.

Each phase of this process is discussed in more detail below.

Selection of Test Programs

The set of test programs used in the MCF evaluation was constrained by budget limitations and the statistical use to be made of the results. Validity of the statistical results required that the programs be a representative set of the kind of operations performed by military computers. Along these lines, it was also considered important that the programs test all significant aspects of the architectures. These considerations would indicate the desirability of a large set of test programs. However, the analysis required that each program be coded frequently enough to allow significant statistical inferrences to be made. Thus budgetary constraints forced a tradeoff between number of tests, length of test, and frequency of coding.

A set of 16 test programs divided into four categories was ultimately selected for the evaluation. The basis of the individual selections was twofold. First, a list of important architectural features was assembled. Features to be tested were.
Interrupt handling and I/O
Executive/ User interaction
Control and branching constructs
Integer arithmetic
Floating Point operations
Character and Bit processing

Addressing mode flexibility
Ability to address large data structures

Second, a set of significant tasks to be performed were considered:
Real time processing
Handling multiple processes
Communications processing
Display processing
Fast table lookup
Packing and Unpacking data
Sorting
Manipulation of list structures
Minimal Difference Search
Character processing

Attempting to maximally cover the two sets above resulted in the selection of the 16 test programs described below.

INTERRUPTS AND TRAPS

0. TTY Input Driver

This is a driver for a simple interrupt driven device. Important characteristics are a low transfer rate (bytes per interrupt), minimal latency from interrupt signal to response, and high flexibility in the nature of the response. These characteristics preclude the use of a typical hardware channel (DMA transfer). The test is typical of a variety of slow speed devices.

1. Message Buffering and Transmission

A high speed DMA device is used to transmit data buffers. The driver's concern is to buffer transmission requests and maintain as high a transfer rate as possible. The computer performs no processing on the data transmitted. This test exercises the channel (DMA) I/O structure of the architecture.

2. Multiple Priority Interrupt Handler

Interrupts from four devices of unequal priority are directed to the appropriate device handlers. The I/O request which is thereby completed is added to the executive's queue so that the appropriate actions may be taken relative to the requesting process. The test performs only the interrupt fielding and request queueing functions. The model is applicable to a variety of real time applications.

3. Virtual Memory Exchange

A protected subroutine facility is provided by a pair of executive calls. The test program performs the memory space and register changes necessary to transfer control. The test measures supervisor call and context swap costs.

MISCELLANEOUS

4. Scale Vector Display

Given a display list and a scale factor, the program produces a scaled display list. The program is a test of integer manipulation and fixed field extraction.

5. Array Manipulation- LU Decomposition

Solution of simultaneous equations using standard Gaussian elimination. Floating point operations, multiple indexing, and nested iteration capabilities are tested.

6. Target Tracking

Given the coordinates of an object, find the closest element to it in a given table. This tests floating point comparison as well as the costs of performing contorted array searches.

7. Digital Communications Processing

This program directs messages to various output lines depending upon their destinations. Fast search and block move capabilities are tested.

ADDRESS MANIPULATION

8. Hash Table Search

The problem is to locate the position a key would occupy in a hash table. This involves address and integer manipulations and indexing.

9. Linked List Insertion

Given a doubly linked list in ascending order, insert a new entry. The test involves pointer extraction and following.

10. Presort on Large Address Space

Manipulate the elements of a very large randomly ordered array to form a partially ordered binary tree. The array is sufficiently large (order 1 Mbyte) that it is necessary to manipulate the page (segment) address registers to access it. This is a test of the cost of randomly addressing a very large address space.

11. Autocorrelate on Large Address Space

This test is complementary to test 10. An autocorrelation is performed on an array large enough to require manipulation of page registers. Floating point and sequential access of large address spaces are tested.

CHARACTER AND BIT MANIPULATION

12. Character Search

A character string is scanned looking for an occurence of a specified string. This program tests character accessing abilities.

13. Boolean Matrix Transpose

This program takes a bit matrix and reflects it about its diagonal. Ability to access and move bits is tested.

14. Record Unpacking

This test program takes an array of tightly packed bit fields and a format string indicating the size of each field and unpacks the fields into another array. The ability to do general field extraction is tested.

15. Vector to Scan Line Conversion

A list of vectors is converted to an equivalent scan line display. This tests bit addressing capabilities as well as some integer manipulations.

Specification and Control of Test Programs

The algorithm to be used in each program was specified in a high order language. The programmers were allowed to make any optimizations that a clever compiler could make, but were not allowed to change the algorithm used. With the exception of the interrupt and trap class of tests, all tests were specified as subroutines; this standardized input/result handling. The calling conventions were specified for each machine in terms of a sample instruction sequence which would produce the machine state to be expected at entry. These steps were used to restrict the variance due to the difference between programmers without restricting their ability to make optimal use of the machine.

Several conventions were adopted with respect to the non- I/O programs. All programs were required to be reentrant. A stack area was supplied on all machines for use by the programmmers. The subroutines were not allowed to alter any data which was on the stack prior to the call, nor were they permitted to leave any items on the stack subsequent to the return. Finally, The subroutines were required to save and restore any processor registers which they altered.

Assignment of Test Programs

Test programmers were assigned test programs in accordance with the statistical design chosen. Each programmer implemented two programs on all five architectures. Pairs of programmers received identical program assignments. The suggested order of writing was different for each programmer to avoid algorithm/machine familiarity interactions. The exact design of the experiment is explained in the analysis section. The assignments are displayed below.

Table 1. Program Assignment

Pgmr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0								*	*							
1						*					*					
2							*			*						
3					*							*				
4				*									*			
5		*													*	
6			*											*		
7	*															*
8	*															*
9			*											*		
10		*													*	
11				*									*			
12					*							*				
13							*			*						
14						*					*					
15								*	*							

Program/Machine in Suggested Order

Pgmr										
0	7/20	8/12	7/19	7/7	7/11	8/11	8/7	8/19	7/12	8/20
1	5/12	10/19	10/20	5/11	5/20	5/19	10/12	10/11	10/7	5/7
2	9/12	6/11	6/20	6/12	9/19	6/19	9/20	6/7	9/11	9/7
3	4/11	4/20	11/19	4/7	11/7	4/19	11/12	11/20	4/12	11/11
4	3/12	3/19	12/12	12/11	12/20	12/19	3/11	3/20	3/7	12/7
5	1/19	14/12	14/19	1/12	1/20	1/11	14/7	14/20	14/11	1/7
6	2/19	2/7	2/12	2/11	13/12	2/20	13/19	13/20	13/7	13/11
7	15/7	0/7	0/20	0/19	0/12	15/20	15/19	0/11	15/12	15/11
8	15/7	15/12	15/11	0/12	0/7	15/19	0/19	15/20	0/20	0/11
9	13/19	2/11	2/20	13/11	2/7	2/19	13/12	13/20	13/7	2/12
10	14/12	1/7	1/19	1/12	14/7	1/11	14/19	14/20	1/20	14/11
11	12/12	3/12	12/20	3/19	12/11	3/20	3/11	3/7	12/19	12/7
12	11/7	11/19	4/7	4/12	4/19	11/12	11/11	4/20	4/11	11/20
13	9/20	9/19	6/20	6/11	9/7	9/12	6/19	6/7	6/12	9/11
14	10/7	10/19	10/20	5/7	5/12	10/12	5/19	10/11	5/11	5/20
15	8/12	8/11	7/11	7/12	7/20	8/20	7/7	8/7	7/19	8/19

Debugging and Execution Testing

An ISPL description of each machine was used to produce an ISPL simulator for each machine[BA76]. Programmers debugged their programs using these simulators. A standard set of test data was defined for each program. A program was defined to be debugged when it could properly execute on this test data. This provides a reasonable assurance of the applicability of the measures obtained without requiring proofs of the correctness of the programs. A subset of this test data was used for evaluation of execution efficiency. The ISPL simulator maintains counters of various memory accesses as well as frequency of execution of each part of the simulator. These counters provided the execution statistics for use in computing the architecture measures.

Measures of an Architectures's Performance

The performance of an architecture on the test programs is measured by the efficiency of the test programs written for that particular architecture. Quantification of the concept of an efficient program allows the comparison of different architectures independant of their implementation. The measures used by the MCF evaluation are such a quantification in terms of space and time.

An efficient program is one which requires a small amount of storage and executes in a short amount of time. Three classes of measures were used to capture this concept. The S measure is a measure of the storage requirements of a program. The M and R measures are measures of execution efficiency.

S MEASURE - TEST PROGRAM SIZE

The S measure is defined as the number of bytes of memory required by the test program. This includes locals allocated on the stack as well as own variables. For the Interrupt and Trap test programs, this also includes memory allocated to interrupt vectors used by the test program. Excluded from the S measure are the parameter block and parameters passed to the routine as well as any global data structures to which the routine has access. This was done to avoid adding a fixed overhead of significant size to each S measure.

A single exception to the parameter exclusion principle was made. Test program 14, Record Unpack, allowed the programmer to chose a representation for the format string. Optimal packing would cause each entry in this string to occupy 6 bits. Because a tradeoff decision between packing efficiency and accessing difficulty was allowed, the size of this parameter was included in the S measure for this program.

For those test programs in which multiple calls were measured, the the stack useage could conceivably vary between calls. The S measure in this case is defined as the maximum of the individual S measures.

EXECUTION EFFICIENCY MEASURES

The time required to execute a given program on a given machine is clearly highly dependant upon the hardware implementation of the machine. An arbitrary architecture can be implemented to execute its instruction set at an arbitrarily fast rate (limited of course by current gate technology). The execution time of a given program is determined by two factors: The amount of processing required, and the rate at which processing is done. The former is dependant upon the program and architecture, the latter upon the hardware implementation. An efficient architecture will minimize the processing required, allowing the most cost effective implementations.

Selection of measures of processing required by a program allows the comparison of the efficiencies of several architectures. Taking instructions as special cases of programs, such measures must, because of the separation into factors assumed above, reflect the differences in execution times of instructions in current implementations. This provides a selection criterion for measures.

Consider the following 3 example instructions selected from the familiar 360/370 architecture.

1.	L	1,0(2)	load from memory
2.	LM	1,6,0(2)	load 6 regs from memory
3.	AE	2,0(2)	floating point add from memory

These examples illustrate two orthogonal factors accounting for the differences in processing required between instructions. Example 2 would be expected to execute more slowly than 1 since it involves reading 5 more words from memory. Memory activity is thus an important factor in execution cost; The M measure was therefore defined as the number of bytes transferred to/from memory. On the other hand, examples 1 and 3 have the same memory activity, but 3 would be expected to execute more slowly. The processor activity involved in floating point operations increases their cost. Processor activity is thus an important factor; This is measured by the R measure. Both M and R are discussed in detail below.

The execution time model used in the MCF evaluation is represented by the following equation:

$$TIME = a*M + b*R$$

Where a and b are constants dependant upon the speed of the memory and processor hardware, respectively. M and R are measures of the processing costs involved in the architecture, independant of implementation.

Measure of Memory Activity - M

An important parameter of a computer system is the bandwidth of its processor/memory interface. Thus a significant determinant of program execution speed is the number of bytes the program transfers to or from memory. The M measure is a measure of memory activity.

The M measure is defined as the number of bytes read or written to main memory during the execution of the test program. Specificly, counting begins at the first instruction of the routine and ends when a return is executed. No activity of the calling routine is counted.

Three M measures were computed. These M measures reflect differences in the width of the memory (and therefore the minimum number of bytes which can be read from a given address). They are refered to as M8, M16, and M32 corresponding to 1, 2, and 4 byte wide memories, respectively.

Certainly, no one would implement the 16 bit machines with 32 bit memories without making some attempt at reasonable utilization of the wider memory. Thus, two adjustments to the M32 definition for the 16 bit machines were made. First, it is assumed that all multiple word references (double integer, floating point, etc.) were aligned on fullword boundaries. This is of course standard practice in most 32 bit machines. Second, the sequential nature of instruction fetch makes it highly desireable to have a 32 bit instruction buffer. Otherwise a sequence of 16 bit instructions would result in each instruction being fetched twice as the low and high halves of the 32 bit word were executed. This implementation was modeled by allowing instruction fetches to fetch 2 bytes, while all other memory accesses must use 4 byte words. These two adjustments define the 32 bit memory system assumed by M32 for the 16 bit machines.

Measure of Processor Activity - R

The activity of the processor during the execution of an instruction is simply the computation of a function. Complexity theory indicates that the cost of this computation can be measured by many step counting functions. Consideration of step counting functions applicable to digital implementations fails to restrict significantly the range of possible cost functions (consider two processors, one which is bit serial, the other uses table lookup in a ROM. Addition is expensive in the former, while all functions are of equal cost in the latter). It is therefore necessary to choose a cost function which represents an implementation that is reasonable given the current state of the art. This is the approach taken in the MCF study.

The R measure for a program is defined as the sum of the R measures for each instruction executed. The R measure of an instruction is defined as the number of CPU cycles required to execute it using the canonical CPU defined below. As for the M measure, no driver activity was included.

Two R measures were computed. One assumed a 16 bit wide ALU as would be used for low performance versions of the UYK7 and GYK12 and most versions of the PDP11,UYK19, and UYK20. The other assumed a 32 bit ALU as would be used for high performance versions of the 11, 19 and 20 and most versions of the 7 and 12. These two measures are refered to as R16 and R32.

MCF CANONICAL CPU

Definition of a reasonable complexity measure for instruction execution neccessitated the choice of a standard structure for the emulating CPU. The structure shown in figure 1 was chosen to be representative of typical medium performance implementations now in use. Data paths and ALU operations reflect the capabilities of current instruction serial hardware units.

Features of the CPU

The CPU includes a register ram, constant rom, temporary latches, as well as a memory address and data register and a parallel ALU. The width of these and the Interconnecting busses is 16 bits for R16 and 32 bits for R32.

The register ram is a standard random access memory used to hold the accumulators, index registers, program counter, and stack pointers of the architecture. During a CPU cycle a single location may be read or written.

The constant rom contains a variety of useful constants for implementing the architecture in question.

The temporary latches are high speed registers used in the interpretation process. They may be read and written on the same cycle.

The ALU is a parallel arithmetic unit capable of integer addition, subtraction and negation, all the standard logical functions such as and and or, as well as n bit shifts and rotates. It is also capable of performing fixed bit substitutions, such as replacing the low byte of one bus with the low byte of the other. The condition codes may be set by its outputs.

Instruction Implementation

Several principles were adopted with respect to the R measure. These were intended to avoid unnecessary complexity. They also avoid arbitrarily penalizing architectures with unique features. Finally, they prevent overheads common to all interpretations from obscuring the differences between instructions. These are outlined below.

1) Instruction decode is excluded. The control operations involved are extremely implementation dependant and represent an unnecessary overhead for the R measure.

2) Memory mapping calculations are presumed to be performed by a separate unit and therefore require no CPU cycles. The activity of the memory map will simply make memory accesses more expensive and therefore is adequately measured by M.

3) The address for a memory access may be obtained from a variety of places such as the MAR, MDR, IR. This eliminates shuffleing operations which are highly implementation dependant.

4) Inter- instruction optimizations are not allowed.

MCF CANONICAL CPU

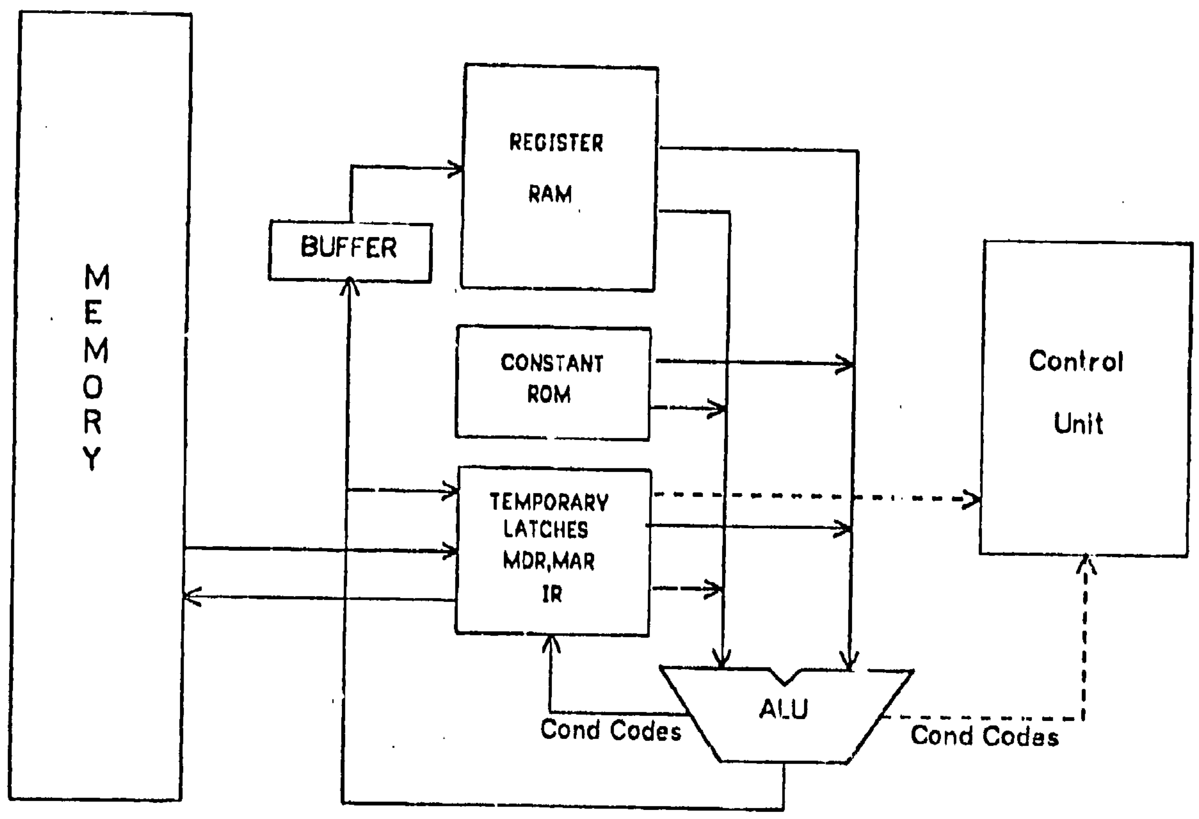


Figure 1

Calculation of R measures

The R measure for each instruction was obtained in two different ways depending upon the complexity of the instruction.

For relatively simple instructions, microcode was generated for the instruction. Adding this to the standard instruction fetch produced the CPU sequence for interpretation and therefore the number of cycles required.
Example 1.

```
R32 for UYK7 instruction  LA    1,0(2)
1. MAR ← INS REG<address> + REG[index 2]
2. REG[acc 1] ← MDR
```

So the R measure is 2 + instruction fetch cost.

Example 2.

```
R32 for GYK12 instruction CMF   2,B
1. TMP1 ← REG[acc 14]
2. TMP2 ← REG[acc 2] and TMP1
3. TMP1 ← MDR and TMP1
4. TMP2 - TMP1 (set CCs)
```

So the R measure is 4 + instruction fetch cost.

Complex instructions such as integer multiply and divide and floating point operations were handled differently. Direct microcoding was deemed undesirable for several reasons. First, generating the microcode would be very time consuming. The effort expended would be far out of line with the accuracy required. Second, the generation of optimal microcode for such instructions is a research problem in itself. Finally, the optimal algorithm, if found, might require a control complexity sufficient to make it impractical. Since all machines would be charged the same basic cost for these operations, we decided to use a reasonable approximation.

The R measure for each of these complex instructions was established by a survey of implementation on current computers. Relative execution times were used to establish an approximate number of CPU cycles required to execute. This computation cost was then added to an operand fetch cost (as determined by microcoding) and an instruction fetch cost to determine the R measure of the instruction.
Example 1.

```
R32 for PDP11  MUL     R1,R2
1. TMP0 ← REG[r1]
2. TMP1 ← REG[r2]
(16 bit multiply computation, 20 cycles)
3. REG[r2] ← TMPRESULT<0:15>
4. REG[r2+1] ← TMPRESULT<16:31>
```

So the r measure is 4 + 20 + instruction fetch cost.

Instruction fetch was assumed to be accomplished by the following microcode:

```
1. MAR,TMP0 ← REG[pc]
```

2. REG[pc] ← TMPO + (instruction size)

where the instruction size was determined by the instruction fetched. Architectures with varying length instructions were allowed to delay the pc increment operation until the instruction size was determined. Unconditional branches were allowed to dispense altogether with cycle 2. Conditional branches assumed no branch occured; cycle 2 was completed. Thus instruction fetch requires 1 cycle for unconditional branches, and 2 for all other instructions.

Statistical Design for the Experiment

The general aim of this experiment is to identify the significant factors which influence the S,M and R measures with emphasis on the significance of the Architecture factor. The primary aim is to associate a quantitative measure with each architecture and to obtain confidence intervals on these measures at some predetermined level of statistical significance (.05). From these we can obtain statistically valid rankings of the architectures. A secondary objective is to obtain information on the variance of programmer ability and interactions of programmers and machines.

The method used to design and analyze the CFA measurements was based on the Analysis of Variance (ANOVA) technique. A rough definition of the technique as given by Scheffe [SC59] is : "ANOVA is a statistical technique for analyzing measurements depending on several kinds of effects operating simultaneously, to decide which kinds of effects are important and to estimate these effects."

The CFA experiment was set up on the assumption that the factors influencing the measurements are 1) Programmers 2) Test Programs and 3) Machines. The measurements to be analyzed are the various S, M and R measures obtained in the course of the experiment. The design involved five machines and sixteen test programs which are hopefully representative of the type of task that these machines would be called upon to handle in normal use. It involves 16 programmers who are assumed to be representative sample from the general population of graduate students at Carnegie- Mellon University. In the most desireable situation, the programmers' prior familiarity with the machines being tested would be uniform across all machines. The proliferation of the PDP11 architecture makes this virtually impossible to obtain. An effort was made to include in the study programmers who had no prior PDP11 experience as well as some who had experience with the NOVA (a UYK19 subset). Thus a secondary goal of the analysis was to determine if programmer familiarity was an important factor.

A complete factorial design would involve each programmer coding each test program on all the machines. This would involve coding 1280 (16*16*5) programs and would give complete information on all the relevant factors and interactions. Attractive as this is, budget consideratjons eliminated this design and a one-eighth fractional design was chosen which involved the coding of a total of 160 programs. In the trade-off we naturally lose some information on certain effects or interactions and obtain only partial information on others. Since our primary goal was to obtain information on the machine effects, a design balanced with respect to machines was chosen. That is, each machine was given the same combinations of the other two factors. Hence 32 programs were coded for each machine. Each programmer was assigned 2 test programs to be done on all 5 machines.

The ANOVA model used for a complete factorial design would be:

$$Y_{ijk} = U+P_i+T_j+M_k+PT_{ij}+TM_{jk}+PM_{ik}+PTM_{ijk}+E_{ijk}$$

where the components are as below.

Y_{ijk}	:	Measurement for the I'th programmer,J'th test program on the K'th machine.
U	:	Grand mean.
P_i	:	Effect of the I'th programmer.
T_j	:	Effect of the J'th test program.
M_k	:	Effect of the K'th machine.
PT_{ij}	:	Effect of the interaction between the I'th programmer and the J'th test program.
TM_{jk}	:	Effect of the intearction between the J'th test program and the K'th machine.
PM_{ik}	:	Effect of the interaction between the I'th programmer and the K'th machine.
PTM_{ijk}	:	Effect of the intearction between the I'th programmer, the J'th test program and the K'th machine.
E_{ijk}	:	A random variable distributed as $N(0,\sigma^2)$

A factor could be of two types - random or fixed . A factor in the design is said to be fixed if our inferences from the experiment are limited to exactly the levels of that factor which were chosen for the experiment. For example, in this design we are interested in comparing only these 5 specific machines and not in comparing them with any random architecture. Hence the Machine factor is a fixed factor and we would be interested in computing the effects of these 5 machines. A 'good' machine (low S,M and R) would have a low machine effect while a 'bad' machine would have a higher machine effect. Since the effects can be calculated to within an additive constant, the constant could be absorbed in the grand mean and unique effects obtained by setting the restriction $\Sigma_k M_k = 0$. This applies to all fixed effects. Now the 'best' machine would have a negative effect while the 'worst' machine would have a positive effect.

On the other hand the programmer and test program factors are random factors in this design. In the programmer case we would not like to limit our universe of inference to the 16 programmers chosen for this study. Instead we assume that these 16 programmers are a random sample from a population of programmers. Each P_i is a random variable with distribution $N(0,\sigma_P^2)$. The σ_P^2 is the variance of the programmer population. Note that $\Sigma_i P_i$ need not be equal to 0. In the random effect case we are interested in the variance of the means (σ_P^2) and not in the expected value of the mean which we have assumed to be 0.

A design like this one which has some of its factors fixed (M) and others random (P and T) is termed a mixed model. The interaction of a fixed and random factor is itself a random interaction. Thus all the interactions in this design are themselves random variables with mean 0 and different variances.

ANOVA models are valid only under certain restrictions on the random error term E_{ijk}. Each E_{ijk} must be normally distributed with mean 0 and variance σ^2. Furthur each E_{ijk} must be independantly normallly distributed (The covariance matrix of the column vector E is σ^2I). In other words the Y_{ijk}'s themselves can be assumed to be random variables with different means, but having the same variance for all i,j and k. One way to check this would be to to actually measure the variance of Y_{ijk}. Unfortunately we cannot estimate the variance as we have only one measurement on Y_{ijk}. We cannot also directly obtain an estimate of the error variance σ^2.

The first obviously impractical solution is to repeat the entire experiment with the same group of programmers assuming that they have had amnesia in between. We would then get slightly different values for the Y_{ijk}'s and from the two sets could check whether the variances of the Y_{ijk}'s are equal and also obtain the variance of the random error term.

This dilemma can be resolved by assuming that certain higher order interactions are negligible and attributing their sums of squares along with their degrees of freedom to the the sum of squares due to error. In this way we obtain an upper bound on σ^2. If the interactions were not actually 0 then we would be overestimating σ^2 and hence being overly conservative about the lengths of our confidence intervals. However this doesn't solve the problem of testing for normality and equal variance. In fact we believe that based on theoretical and intuitive considerations that the variance of the measures are not equal. We furthur postulate that the the standard deviation of any Y_{ijk} is directly proportional to the mean of Y_{ijk} and since the means are not equal, neither are the variances.The hypothesis that the standard deviation is directly proportional to the mean was validated by grouping the 160 data points for each measure into 80 pairs. A pair consists of the measures for the same test program on the same machine but coded by different programmers. The assumption made here is that differences between programmers are not pronounced and inso far as that assumption is wrong,we obtain a crude estimate of the mean and variance for each pair. The estimates are bound to be noisy as we are computing them from just 2 elements. A scatter plot of Log Variance was plotted against Log Mean and a straight line was fitted by the least squares method. The slope of the line was around 2 in all the cases which indicates that the variance is proportional to the square of the mean and hence that the standard deviation was proportional to the mean.The plots are shown in Appendix 2. An appropriate transformation of the data would equalize the variance approximately [SC59]. Since the std. deviation is proportional to the mean the appropriate variance stabilizing transform is the LOG transform. The ANOVA assumptions will be met if we model Log(Y_{ijk}) as an additive model.

$$\log Y_{ijk} = U+P_i+T_j+M_k+PT_{ij}+TM_{jk}+PM_{ik}+PTM_{ijk}+E_{ijk}$$

Exponentiating both sides we get the intuitively attractive multiplicative model:

$$Y_{ijk} = u*p_i*t_j*m_k*pt_{ij}*tm_{jk}*pm_{ik}*ptm_{ijk}*e_{ijk}$$

where the relation between the lower case and upper care varaibles is $U = \log(u)$ and so on. The conditions on each factor will of course be changed. (For example $\Pi_k M_k = 1$). The 'best' machine would have a multiplicative effect less than 1 while the 'worst' machine would have one greater than one. The significance of the multiplicative effect can be best shown by an example. If the multiplicative effect is 0.81 for the S measure on the PDP-11, it would indicate that the the PDP-11 takes 81% (on the average) of the static storage that a hypothetical average machine would take for executing a random program.

Nested factorial designs are those in which not all factors are crossed with every other factor. A factor could instead be nested within another. Our design would be split into 2 phases of 80 data points each. The first phase would consist of the data from programmers '0' through '7' and the second would be the data from programmers '8' through '15'. Taking either half as an example we note that every level of the test program factor appears together with only a single level of the programmer factor. In other words the test programs that a programmer does are distinct from those done by any other programmer in his half. Thus the test program factor is nested within the programmer factor and hence we have no interaction between programmer and test program. In our notation the factors corresponding to the parenthisized subscripts have nested within them the factor corresponding to the next non parenthized subscript. For example $T_{(i)j}$ would correspond to the effect of the J'th (J=1 or 2) program of the I'th programmer. The subscript 'j' will thus always appear associated with a parenthesized 'i'. Hence the transformed model for the nested factorial design (first half) would be:

$$Y_{ijk} = U + P_i + T_{(i)j} + M_k + PM_{ik} + TM_{(i)jk} + E_{ijk}$$

where Y_{ijk} is the log of S,M or R_{ijk} and the range of the subscripts are as follows: $i=1:I, j=1:J, k=1:K$ where $I=8$, $J=2$ and $K=5$. The corresponding multiplicative model is obtained by exponentiating both sides.

The restrictions on the variables are:
Expected values of P_i, $T_{(i)j}$, PM_{ik}, $TM_{(i)jk}$ and E_{ijk} are 0.
Corresponding variances are σ_P^2, σ_T^2, σ_{PM}^2, σ_{TM}^2, σ^2.
Furthur $\Sigma_k M_k = \Sigma_k PM_{ik} = \Sigma_k TM_{(i)jk} = 0$.
We define $\sigma_{PM}^2 = \Sigma_k \sigma_{PM,k}^2 / (K-1)$ and

$$\sigma_{TM}^2 = \Sigma_k \sigma_{TM,k}^2 / (K-1). \qquad [1]$$

The total sum of squares $\Sigma_i \Sigma_j \Sigma_k (Y_{ijk}\text{-mean})^2$ is then decomposed into the sums of squares due to each component according to the formulae given in Appendix 1. The corresponding mean squares are obtained by dividing the sums of squares by their degrees of freedom. Theoretically expected values for the Mean squares are given in Table 2. The only mean values that we are interested in calculating are the M_k's which are computed as:

$M_k = Y_{..k} - Y_{...}$ where the dot notation denotes that an average is taken over the dotted subscripts.

Comparisions of the machine effects would be more useful rather than the absolute values of the M_k's. Confidence intervals for the differances of the machine effects(contrasts) are estimable. The mean value for the statistic $Y_{..l}-Y_{..m}$ is the contast between machine 'l' and machine 'm' or M_l-M_m. The variance of this contrast depends on our universe of inference. If all the factors are fixed then the variance of the contrast is $2\sigma^2/IJ$ where σ^2 is the variance of the error term. However if the programmers and test programs are taken as random effects then the variance is $2(\sigma^2+\sigma_{TM}{}^2+J\sigma_{PM}{}^2)/IJ$. The variance is larger under these assumptions and hence the confidence intervals are also larger. The two tailed T test can then be used to determine the confidence intervals for the contrast. For example if $\sigma_C{}^2$ is the estimated variance of the contrast with estimated mean μ_C, then the interval for the true mean is:

$\mu_C-t(df,.025)\sigma_C <= \mu_C <= \mu_C+t(df,.975)\sigma_C$ with 95% confidence. 'df' is the number of degrees of freedom with which the error variance is computed (41 in our case).

Instead of assuming that the third order interactions are negligible, we could look at the complete design as a 1/8 fraction of a complete 2^8*5 design. The programmer and test program factors are each represented by 4 pseudo-factors at 2 levels each. The model assumed is

$$Y = XB + E$$

where Y and E are 160 length column vectors.[CO61] The parameter to be estimated is the B column vector. E is a vector of the random error variables with mean 0 and having a covariance matrix of $\sigma^2 I$. The number of parameters fitted must be less than 160 or we would get a perfect fit. Instead 119 parameters are fitted leaving 41 degrees of freedom to estimate the error. The X array is a 160 by 119 array of the appropriate orthogonal polynomials.[CO61] The parameters not estimated are the fifth or higher order interactions of the pseudo-factors.

<u>ASSIGNMENT OF PROGRAMS</u>: The main problem is the choice of which treatment combinations are to be included in the fractional design. We must to choose 32 combinations of the 256 possible combinations of programmers and test programs. These combinations are of course replicated for all 5 machines to maintain balance. The key is to choose the combinations such that the effects and interactions which are confounded are the ones which are of little interest. Let A,B,C and D be the pseudo-factors corresponding to the test program factor and E,F,G and H to the programmer factor. Following the procedure outlined in [CO61] we select 3 relatively unimportant interactions to be confounded. The 4 generalized interactions are generated from these three. In general it would be a good idea to confound the higher order intearctions, but care must be taken in choosing the 3 basic interactions as the generalized interactions may be confounding main effects. Another restriction enforced by the need for balance is that each of the interactions confounded must have the same number of pseudo-factors from each main factor.

The three basic interactions to be confounded were chosen to be:

ABEF = ADEG = ABCDEFGH

The generalized interactions (which are aliases of the basic interactions) are to be obtained by multiplying together any combination of the basic interactions with the squared terms replaced by unity [AN73] (on account of the 2 levels of each pseudofactor). The seven interactions confounded with the grand mean are:

I=ABEF=ADEG=ABCDEFGH=BDFG=CDGH=BCFH=ACEH where I denotes the grand mean.

There is a complete loss of information on these interactions and there is partial loss of information on many of the other factors and interactions. For example to find the interactions confounded with A, we just multiply(index modulo 2) the above equation by A and obtain:

A=BEF=DEG=BCDEFGH=ABDFG=ACDGH=ABCFH=CEH.

We note that the main effect A is confounded only with third or higher order interactions, but it must be remembered that interactions among the pseudofactors could actually be a main effect for an original factor. For example the third order interaction of the pseudo-factors A,B and C is actually part of the main effect of the test program factor which is made up of the 4 pseudo-factors, the 6 two pseudo-factor interactions, the 4 third order and 1 fourth order interaction making it a combination of 15 effects. This choice seems to be optimal (within renaming the variables) under the restrictions of a balanced design to ensure that machine effects are unconfounded and the budget constraints that force us to take a 1/8 fraction, to minimize the confounding problem.

The 3 defining equations to determine which 32 of the 256 observations to take [CO61] are obtained from the basic confounded interactions. They are :

$x_1+x_2+x_5+x_6 = 1 \pmod 2$
$x_1+x_4+x_5+x_7 = 1 \pmod 2$
$x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8 = 1 \pmod 2$

where each x_i is 0 or 1 to denote the 2 levels of the corresponding pseudo-factor.

There are 32 sets of solutions for these equations. As an example one such solution set would be $x_1x_2x_3x_4 = 0000$ and $x_5x_6x_7x_8 = 0111$. We can denote the test programs and the programmers by the numbers 0 through 15 base 2. The 32 solution sets then give us the assignments to be made. The example solution assigns test program '0' to programmer '7'. The assignments to each programmer were summarized in Table 1. Note that the design is very symmetrical and has pairs of programmers assigned the same set of programs.

The X matrix is generated iteratively with the columns due to the main effects of the pseudo-factors and the linear,quadratic,cubic and quartic effects of the machine factor being inserted first. We then append the columns due to the interactions (calculated by term by term multiplication of the appropriate columns). We must make sure at each stage that we do not append a column for an interaction whose alias has already appeared as this would destroy the linear independance of the columns.[CO61] We stop when we get to the 5 factor interactions which gives us an X matrix of size

160 by 119. The Y column vector contains the log of the measurements(S,M or R). Let B be the vector of parameters which will minimize the sum of the squares of the difference between the Y values and the predicted Y values. Then the expected value of B is B and is given by:

$B = (X'X)^{-1} X'Y$ if $X'X$ is not singular. Interestingly enough $X'X$ is a diagonal matrix and is easy to invert. The sum of squares due to error is $SSE = Y'Y - B'X'Y$ with 41 degrees of freedom. An estimate of the error variance σ^2 is then obtained by dividing the SSE by the degrees of freedom. We can also estimate the goodness of the fit by doing an F test on the quotient of the mean squares due to regression and error.

We obtain estimates of the variance components σ^2, σ_T^2, σ_{PM}^2, σ_{TM}^2 and σ_M^2. σ_M^2 is not really a variance at all but is given by $(\Sigma_k M_k^2)/(K-1)$ which has the same general form as a variance. The estimates are themselves obtained as linear combinations of some of the expected mean squares except for σ_T^2 and σ_{TM}^2. These can be estimated only if we have an independant estimate of the error variance σ^2. The σ^2 is obtained from the fractional factorial analysis. There is a non-zero probablity that any of these estimates could be negative. [SC59] Since the variables estimated are non-negative (being variances) the conclusion drawn is that the negative estimates are actually estimating a variance very close to 0. The estimate of σ_{PM}^2 is negative and hence was assumed to be 0. It must be noted that these estimates of the variances could have very wide confidence intervals and are hence not as reliable as the estimates of the machine effects. They do however give us rough estimates of these interesting parameters. It also gives us the contributions of each factor or interaction to the variance of a data point. As expected most of the variation is due to the test program factor with the programmer factor being next in importance. The variations due to the machine and interaction of test programs and machines are smaller, but significant. Since σ_{PM}^2 was estimated to be 0 in all the cases, the inference is that programmer familiarity with machines was not very important. The estimated values for the variance are shown in Table 2. Assuming a $N(0,\sigma_P^2)$ distribution for P_i we can conclude that 68% of the programmers have scores lying between σ_P and $-\sigma_P$. In terms of the more appealing multiplicative model the interpretation is that 68% of the programmers have scores lying between $\exp(-\sigma_P)$ and $\exp(\sigma_P)$. For example in the S measure where $\sigma_P^2=0.0435$, 68% of the programmers lie between 0.812 and 1.232. Due to the fact that the distribution is log-normal the mean is not 1 but $\exp(\sigma_P/2)$ which is close to 1 for small σ_P. The average programmer then would score 1.00 (1.02 accurately) while 68% of the programmers would have a score between 0.81 and 1.23. These results are shown in Table 2 The results for the two parts of the nested factorial experiment were averaged with equal weightage and the results for the various estimates of the parameters in the model are shown in Table 3

Machine effects can also be obtained for certain interesting subsets of the test programs. The corresponding confidence intervals widen as a consequence of the smaller number of data points that the used to estimate the machine effects. Machine effects were obtained for the following subsets:

Traps and Interrupts	(Test Programs 0,1,2,3)
Miscellaneous	(Test programs 4,5,6,7)

Address Manipulation	(Test programs 8,9,10,11)
Character and Bit manipulation	(Test programs 12,13,14,15)
Supervisor programs	(Test Programs 0,1,2,3,10,11)

The results for the subsets are shown in Table 3

Results

The results of the statistical analysis are displayed in tabular and graph form for six groupings of test programs. The groups are: All programs, the four subgroups(Interrupt and Trap, Miscellaneous, Address Manipulation, Character and Bit Manipulation), executive mode programs (Interrupt and Trap as well as those which manipulate page registers), and user mode programs.

ALL TEST PROGRAMS

The results from the group of all programs are the most statistically significant (have the smallest confidence intervals). Looking at the S measure we find the 16 bit machines make up the best group, with the PDP11 significantly better than the UYK19. The GYK12 and UYK7, in that order, make up the worst group. This split is due to the availability of 2 byte instructions to perform common operations on the 16 bit machines. The 32 bit machines require the use of 4 byte instructions for the same operations. The UYK7 from the latter group does in fact allow 2 byte instructions; however they must occur in pairs. This results in a large number of 2 byte NOPs as well as obscure coding. The UYK7 also has an addressing structure ill suited to anything other than absolute addressing. This causes its general performance to be poor.

In the M measures the UYK20 and GYK12 both move up relative to the others. These machines have very similar data operations (16 registers, register-register and register-memory operations, similar addressing). The UYK20 utilizes the frequent occurence of small constants by providing short literal and memory reference instructions, as well as short register-register instructions. The GYK12 instructions are all 4 bytes long. We believe this to be the primary reason for their difference in performance.

The PDP11 drops significantly behind the UYK20 in the M measure. This is probably due to a combination of fewer registers (6 vs 16 useable) and a lack of short literal operations.

The UYK19 does quite poorly on M and R. This deficiency arises as a result of its few registers (4, only 2 useable for indexing) and is aggravated by instruction set restrictions. The original instruction set (that of the NOVA) consumed a great deal of the 16 bit instruction space. As a result, the instructions added by ROLM were extremely limited in terms of operand fields. This resulted in further restriction of the register utilization flexibility. The register restrictions prevent code motion optimizations which would move instructions out of loops by precomputing values and saving them in registers.

The separation of accumulators and index registers in the UYK7 seems to preclude its gaining any advantage from its large number of registers.

The R measure (which is isolated from average instruction size) shows the UYK20 and GYK12 clustered at the top. Since the GYK12 is a 32 bit machine, it is at somewhat of a disadvantage in R16, but R32 puts it on top by a sizeable margin.

The R results for the PDP11 indicate that the 11's addressing modes generate a computational burden somewhat greater than those of more conventional machines.

The overall results of this experiment thus show the UYK20 to be at or near the top on all measures, surpassed only by the PDP11 in program size and the GYK12 in high performance computational costs.

SUBGROUP ANALYSES

The most outstanding of the subgroup results are from the Interrupt and Trap group. The GYK12 moves into first place for all measures except S. The advantages of the GYK12 level structure in this area are sufficient to offset the disadvantages of the wide instructions.

The UYK20 falls dramatically in S measure in this group, and loses its M measure advantages over the PDP11. An examination of the individual test program results reveals test 2 (Priority Interrupts) to be the problem. The UYK20 has two weaknesses in this group:

1. The interrupt structure of the UYK20 is very poor. Any attempt to impose a priority structure on devices results in monumental overheads.

2. The UYK20 provides NO kernel/user separation or protection (This was one reason the CFA committee in Phase I eliminated the UYK20 from consideration as a future military standard architecture).

It is noted in passing that the UYK7 also performs abysmally on this group of test programs.

The Character and Bit manipulation tests indicate an advantage for character addressable machines (PDP11, UYK20). Also the bit field extraction facilities of the UYK7 make a significant improvement in its performance, especially in R.

SUMMARY

The significant properties of the machines tested are summarized below. Critical points are indicated by >.

PDP11

1. Byte addressing is advantageous.
2. Overall second in performance.
3. Addressing modes increase computational costs.

UYK20

1. Overall first in performance.
2. Short instructions advantageous.
3. Byte addressing helpful.
4.>Very poor interrupt structure.
5.>No kernel/user protection.

UYK7

1. Bit extraction useful.
2. Separate accumulator/index registers increase costs.
3. Wide instructions memory inefficient; short ones difficult to use.
4.>Poor addressing structure.
5.>Poor interrupt structure.

GYK12

1. Level structure advantageous in interrupt handling.
2. Wide instructions memory inefficient.

UYK19

1. Few registers results in poor execution performance.
2. Instruction encoding restri[illegible] operand accessing flexibility.

Acknowedgements

We wish to acknowledge the considerable efforts of Paul Shaman in the statistical design and analysis of this study.

References

[AN74] Design of Experiments-A Realistic Approach, V.L. Anderson and R.A.Maclean, Michael Dekker, NY, 1974.

[BA76] Barbacci, M. R., The Symbolic Manipuation of Computer Descriptions: ISPL Compiler and Simulator, Technical Report, Deparlment of Computer Science, Carnegie-Mellon University, 1976.

[CO61] Fractional Factorial Designs for experiments with factors at two and three levels, W.S.Connor and Shirley Young, National Bureau of Standards-Applied Math Series-58 , Sept. 1, 1961.

[FU76] Fuller, S.F., W. E. Burr, P. Shaman, D. Lamb: Evaluation of Computer Architectures Via Test programs. Volume III of Computer Family Architecture Selection Committee Final Report, Naval Research Laboratory, Washington, D.C., 1-Dec-1976.

[MC46] Military Computer Family, Selection Methods for a Computer Family Architecture. Reprinted from AFIPS-Conference Proceedings, Volume 46, AFIPS Press, Montvale, N.J.

[SC59] The Analysis of Variance, Henry Scheffe, John Wiley and Sons,Inc, 1959.

TABLE 2

Measure	σ_P^2	σ_T^2	σ_M^2	σ_{TM}^2	σ^2
Log S	.0435	.0946	.0366	.0236	.0641
Log M_8	.1948	.6986	.0631	.0425	.1115
Log M_{16}	.1917	.7046	.0634	.0433	.1065
Log M_{32}	.1917	.6473	.0444	.0448	.1077
Log R_{16}	.3164	1.031	.0273	.0492	.1055
Log R_{32}	.3039	.9400	.0462	.0429	.1024

In all cases σ_{PM}^2 was negative implying that $\sigma_{PM}^2 = 0$.

Measure	Mean	Range of Scores over which 68% of the programmers lie.
S	1.022	.812 to 1.232
M_8	.1.102	.643 to 1.555
M_{16}	1.100	.645 to 1.549
M_{32}	1.100	.645 to 1.549
R_{16}	1.171	.570 to 1.755
R_{32}	1.164	.576 to 1.735

TABLE 3 - ADDITIVE MACHINE EFFECTS

All Test Programs

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19	CI-95% (FIXED)	CI-95% (RANDOM)
LOG(S) :	-0.200	-0.120	0.261	0.135	-0.076	0.128	0.150
LOG(M8) :	-0.130	-0.318	0.324	-0.042	0.166	0.169	0.198
LOG(M16):	-0.126	-0.317	0.332	-0.047	0.158	0.165	0.195
LOG(M32):	-0.010	-0.228	0.162	-0.182	0.257	0.166	0.197
LOG(R16):	0.032	-0.264	0.113	-0.037	0.156	0.164	0.199
LOG(R32):	0.142	-0.145	-0.063	-0.229	0.295	0.162	0.192

Interrupts and Traps - Programs 0-3

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19	CI-95% (FIXED)	CI-95% (RANDOM)
LOG(S) :	-0.239	0.131	0.408	-0.134	-0.166	0.256	0.299
LOG(M8) :	-0.135	-0.136	0.572	-0.272	-0.029	0.337	0.396
LOG(M16):	-0.127	-0.128	0.569	-0.277	-0.038	0.330	0.391
LOG(M32):	0.001	-0.005	0.329	-0.437	0.112	0.331	0.394
LOG(R16):	0.066	-0.389	0.524	-0.356	0.155	0.328	0.397
LOG(R32):	0.160	-0.289	0.361	-0.486	0.255	0.323	0.385

Miscellaneous - Programs 4-7

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19	CI-95% (FIXED)	CI-95% (RANDOM)
LOG(S) :	-0.218	-0.223	0.268	0.161	0.013	0.256	0.299
LOG(M8) :	-0.211	-0.438	0.324	0.029	0.296	0.337	0.396
LOG(M16):	-0.214	-0.441	0.33[illegible]	0.026	0.293	0.330	0.391
LOG(M32):	-0.086	-0.341	0.183	-0.121	0.365	0.331	0.394
LOG(R16):	-0.008	-0.207	0.087	0.105	0.024	0.328	0.397
LOG(R32):	0.076	-0.081	-0.093	-0.089	0.186	0.323	0.385

Address Manipulation - Programs 8-11

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19	CI-95% (FIXED)	CI-95% (RANDOM)
LOG(S) :	-0.134	-0.122	0.173	0.207	-0.124	0.256	0.299
LOG(M8) :	-0.001	-0.260	0.133	0.086	0.041	0.337	0.396
LOG(M16):	0.000	-0.260	0.133	0.086	0.041	0.330	0.391
LOG(M32):	0.117	-0.182	-0.007	-0.043	0.115	0.331	0.394
LOG(R16):	0.111	-0.123	-0.112	0.068	0.056	0.328	0.397
LOG(R32):	0.224	-0.013	-0.278	-0.139	0.205	0.323	0.385

Character and Bit Manipulation - Programs 12-15

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19	CI-95% (FIXED)	CI-95% (RANDOM
LOG(S) :	-0.210	-0.266	0.195	0.306	-0.025	0.256	0.299
LOG(M8) :	-0.175	-0.438	0.268	-0.010	0.354	0.337	0.396
LOG(M16):	-0.164	-0.439	0.292	-0.026	0.337	0.330	0.391
LOG(M32):	-0.073	-0.383	0.143	-0.125	0.438	0.331	0.394
LOG(R16):	-0.041	-0.338	-0.046	0.037	0.388	0.328	0.397
LOG(R32):	0.106	-0.199	-0.241	-0.201	0.535	0.323	0.385

Executive Mode - Programs 0-3,10,11

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19	CI-95% (FIXED)	CI-95% (RANDOM)
LOG(S) :	-0.189	0.039	0.309	-0.050	-0.108	0.209	0.244
LOG(M8) :	-0.044	-0.196	0.346	-0.182	0.077	0.275	0.324
LOG(M16):	-0.039	-0.190	0.344	-0.185	0.070	0.269	0.319
LOG(M32):	0.086	-0.084	0.146	-0.330	0.183	0.271	0.322
LOG(R16):	0.124	-0.287	0.245	-0.212	0.129	0.268	0.324
LOG(R32):	0.225	-0.184	0.073	-0.371	0.258	0.264	0.314

User Mode - Programs 4-9,12-15

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19	CI-95% (FIXED)	CI-95% (RANDOM)
LOG(S) :	-0.207	-0.215	0.232	0.246	-0.056	0.162	0.189
LOG(M8) :	-0.182	-0.391	0.312	0.043	0.219	0.213	0.251
LOG(M16):	-0.179	-0.393	0.325	0.035	0.211	0.208	0.247
LOG(M32):	-0.068	-0.314	0.172	-0.092	0.302	0.210	0.249
LOG(R16):	-0.023	-0.251	0.034	0.069	0.172	0.207	0.251
LOG(R32):	0.092	-0.122	-0.144	-0.143	0.318	0.204	0.243

MULTIPLICATIVE MACHINE EFFECTS

All Test Programs

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19
MACHINE EFFECTS LOG(S) :	0.819	0.887	1.298	1.144	0.927
MACHINE EFFECTS LOG(M8) :	0.878	0.728	1.383	0.959	1.180
MACHINE EFFECTS LOG(M16):	0.881	0.728	1.394	0.954	1.171
MACHINE EFFECTS LOG(M32):	0.990	0.796	1.176	0.834	1.294
MACHINE EFFECTS LOG(R16):	1.033	0.768	1.120	0.964	1.169
MACHINE EFFECTS LOG(R32):	1.152	0.865	0.939	0.796	1.343

Interrupts and Traps - Programs 0-3

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19
MACHINE EFFECTS LOG(S) :	0.787	1.140	1.503	0.875	0.847
MACHINE EFFECTS LOG(M8) :	0.874	0.872	1.772	0.762	0.972
MACHINE EFFECTS LOG(M16):	0.881	0.880	1.767	0.758	0.963
MACHINE EFFECTS LOG(M32):	1.001	0.995	1.390	0.646	1.118
MACHINE EFFECTS LOG(R16):	1.069	0.678	1.689	0.700	1.167
MACHINE EFFECTS LOG(R32):	1.174	0.749	1.434	0.615	1.290

Miscellaneous - Programs 4-7

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19
MACHINE EFFECTS LOG(S) :	0.804	0.800	1.307	1.174	1.013
MACHINE EFFECTS LOG(M8) :	0.810	0.645	1.383	1.030	1.344
MACHINE EFFECTS LOG(M16):	0.808	0.643	1.399	1.027	1.340
MACHINE EFFECTS LOG(M32):	0.918	0.711	1.201	0.886	1.440
MACHINE EFFECTS LOG(R16):	0.992	0.813	1.090	1.110	1.024
MACHINE EFFECTS LOG(R32):	1.079	0.923	0.911	0.915	1.205

Address Manipulation - Programs 8-11

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19
MACHINE EFFECTS LOG(S) :	0.875	0.885	1.189	1.230	0.883
MACHINE EFFECTS LOG(M8) :	0.999	0.771	1.142	1.090	1.042
MACHINE EFFECTS LOG(M16):	1.000	0.771	1.142	1.090	1.042
MACHINE EFFECTS LOG(M32):	1.124	0.834	0.993	0.958	1.122
MACHINE EFFECTS LOG(R16):	1.118	0.885	0.894	1.070	1.058
MACHINE EFFECTS LOG(R32):	1.251	0.987	0.758	0.870	1.228

Character and Bit Manipulation - Programs 12-15

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19
MACHINE EFFECTS LOG(S) :	0.811	0.766	1.215	1.358	0.975
MACHINE EFFECTS LOG(M8) :	0.840	0.645	1.308	0.990	1.425
MACHINE EFFECTS LOG(M16):	0.849	0.645	1.339	0.975	1.401
MACHINE EFFECTS LOG(M32):	0.929	0.682	1.154	0.883	1.549
MACHINE EFFECTS LOG(R16):	0.960	0.713	0.955	1.038	1.474
MACHINE EFFECTS LOG(R32):	1.112	0.820	0.786	0.818	1.707

Executive Mode - Programs 0-3,10,11

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19
MACHINE EFFECTS LOG(S) :	0.827	1.039	1.362	0.951	0.898
MACHINE EFFECTS LOG(M8) :	0.957	0.822	1.413	0.834	1.080
MACHINE EFFECTS LOG(M16):	0.962	0.827	1.411	0.831	1.073
MACHINE EFFECTS LOG(M32):	1.090	0.919	1.157	0.719	1.200
MACHINE EFFECTS LOG(R16):	1.132	0.750	1.278	0.809	1.138
MACHINE EFFECTS LOG(R32):	1.252	0.832	1.076	0.690	1.294

User Mode - Programs 4-9,12-15

	PDP-11	UYK-20	UYK-7	GYK-12	UYK-19
MACHINE EFFECTS LOG(S) :	0.813	0.806	1.261	1.279	0.945
MACHINE EFFECTS LOG(M8) :	0.834	0.676	1.366	1.044	1.245
MACHINE EFFECTS LOG(M16):	0.836	0.675	1.385	1.036	1.235
MACHINE EFFECTS LOG(M32):	0.934	0.731	1.188	0.912	1.353
MACHINE EFFECTS LOG(R16):	0.977	0.778	1.034	1.071	1.187
MACHINE EFFECTS LOG(R32):	1.096	0.885	0.866	0.867	1.374

APPENDIX 1

$Y_{ijk} = U + P_i + T_{(i)j} + M_k + PM_{ik} + TM_{(i)jk} + E_{ijk}$
Range of the subscripts are : $i = 1{:}I$, $j = 1{:}J$, $k = 1{:}K$
where $I = 8$, $J = 2$ and $K = 5$

		Deg. of Freedom
SS_P	$= JK\,\Sigma_i\,(Y_{i..} - Y_{...})^2$	$I-1$
SS_T	$= K\,\Sigma_i \Sigma_j\,(Y_{ij.} - Y_{i..})^2$	$I(J-1)$
SS_M	$= IJ\,\Sigma_k\,(Y_{..k} - Y_{...})^2$	$K-1$
SS_{PM}	$= J\,\Sigma_i \Sigma_k\,(Y_{i.k} - Y_{i..} - Y_{..k} + Y_{...})^2$	$(I-1)(K-1)$
SS_{TM}	$= \Sigma_i \Sigma_j \Sigma_k\,(Y_{ijk} - Y_{ij.} - Y_{i.k} + Y_{i..})^2$	$I(J-1)(K-1)$

Theoretical Expected values of the mean squares obtained by dividing the corresponding sums of squares by their degrees of freedom. The analysis assumes that the P and T factors are random and the machine or M factor is fixed.

$E(MS_P) = \sigma^2 + K\sigma_T^2 + JK\sigma_P^2$

$E(MS_T) = \sigma^2 + K\sigma_T^2$

$E(MS_M) = \sigma^2 + \sigma_{TM}^2 + J\sigma_{PM}^2 + IJ\sigma_M^2$

$E(MS_{PM}) = \sigma^2 + \sigma_{TM}^2 + J\sigma_{PM}^2$

$E(MS_{TM}) = \sigma^2 + \sigma_{TM}^2$

The estimates of the variances are calculated as below,

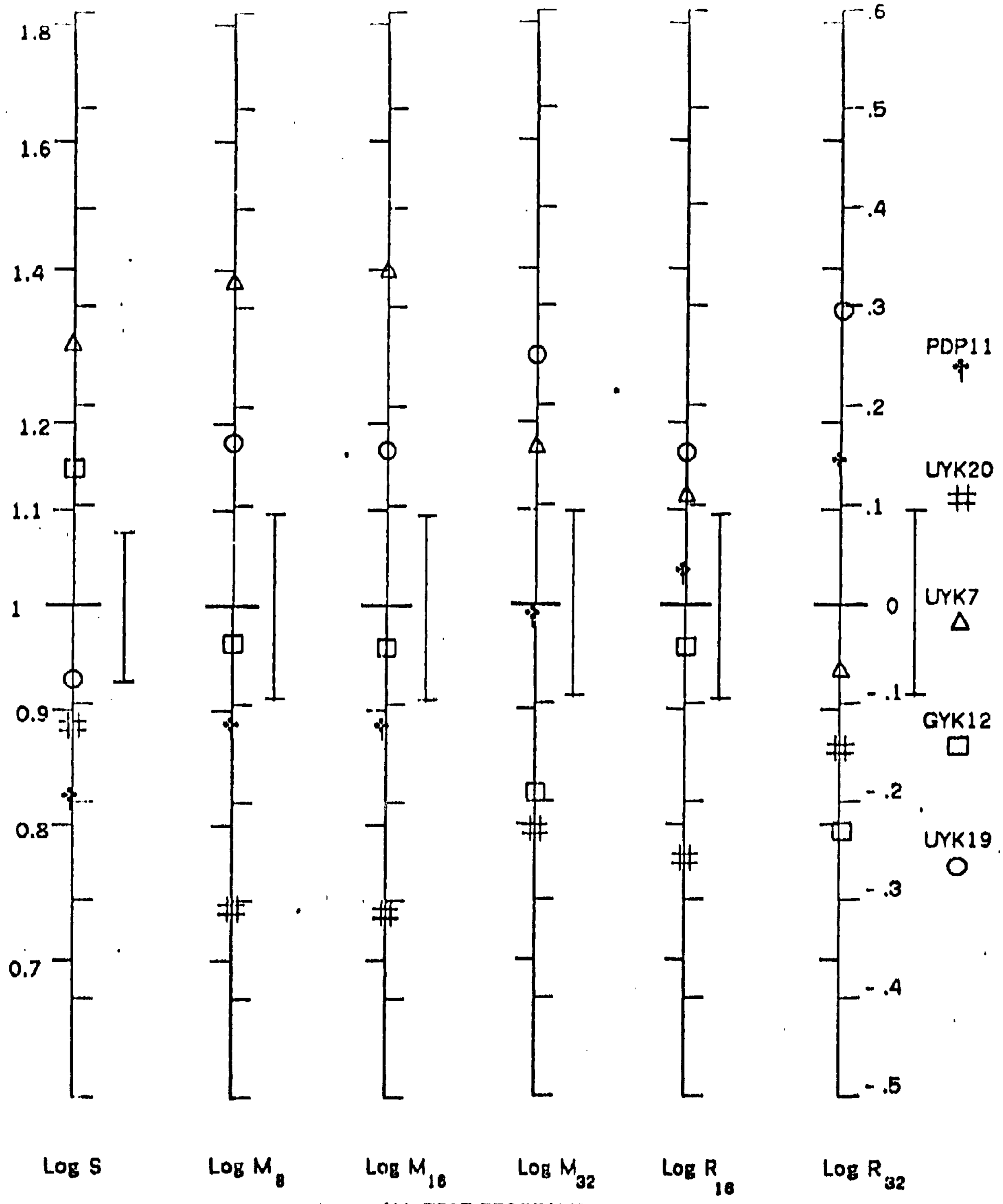
$\sigma_P^2 = (\ E(MS_P) - E(MS_T)\ )\ /\ JK$

$\sigma_T^2 = (\ E(MS_T) - \sigma^2\ )\ /\ K$

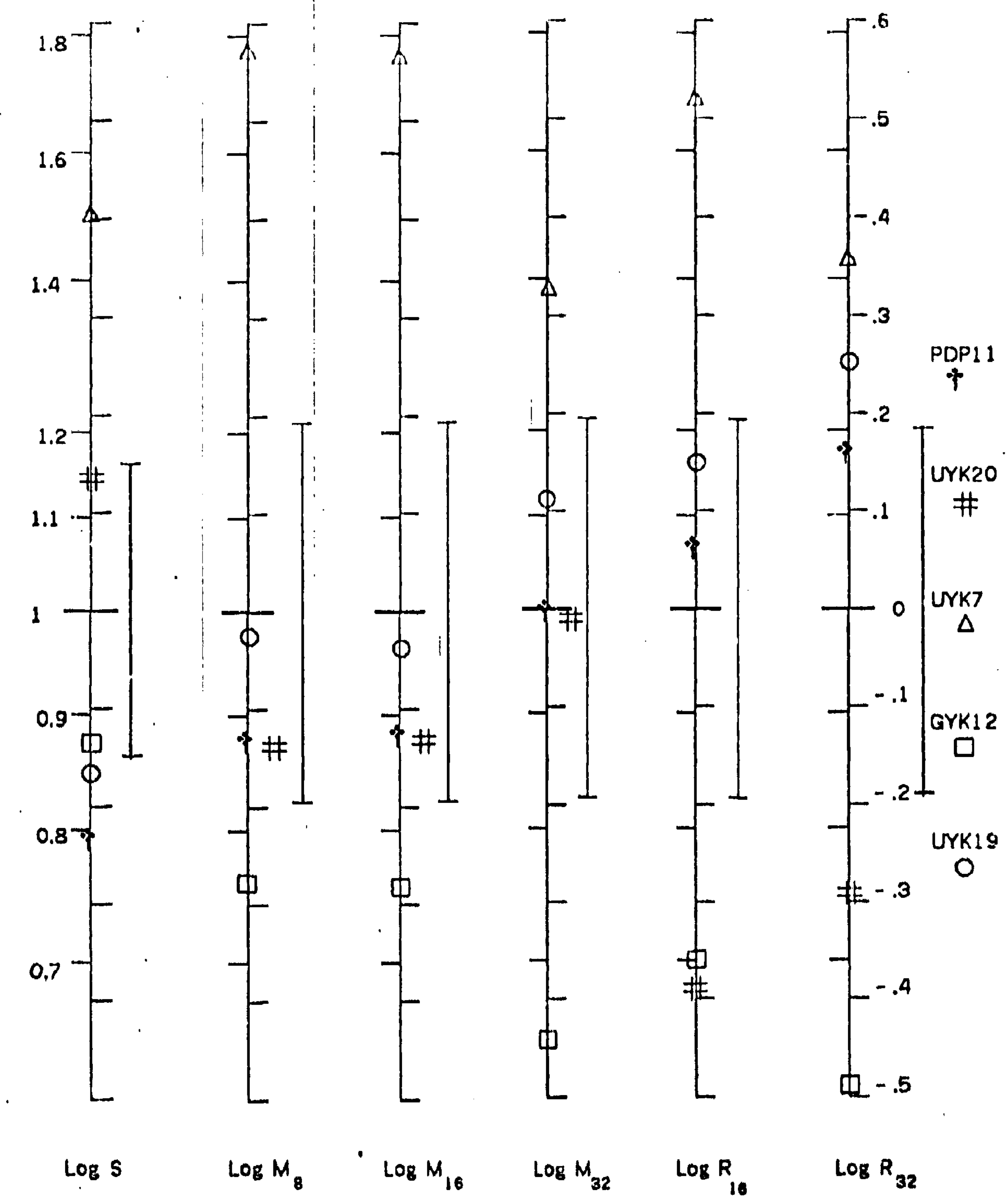
$\sigma_{PM}^2 = (\ E(MS_{PM} - E(MS_{TM})\ )\ /\ J$

$\sigma_{TM}^2 = (\ E(MS_{TM}) - \sigma^2\ )$

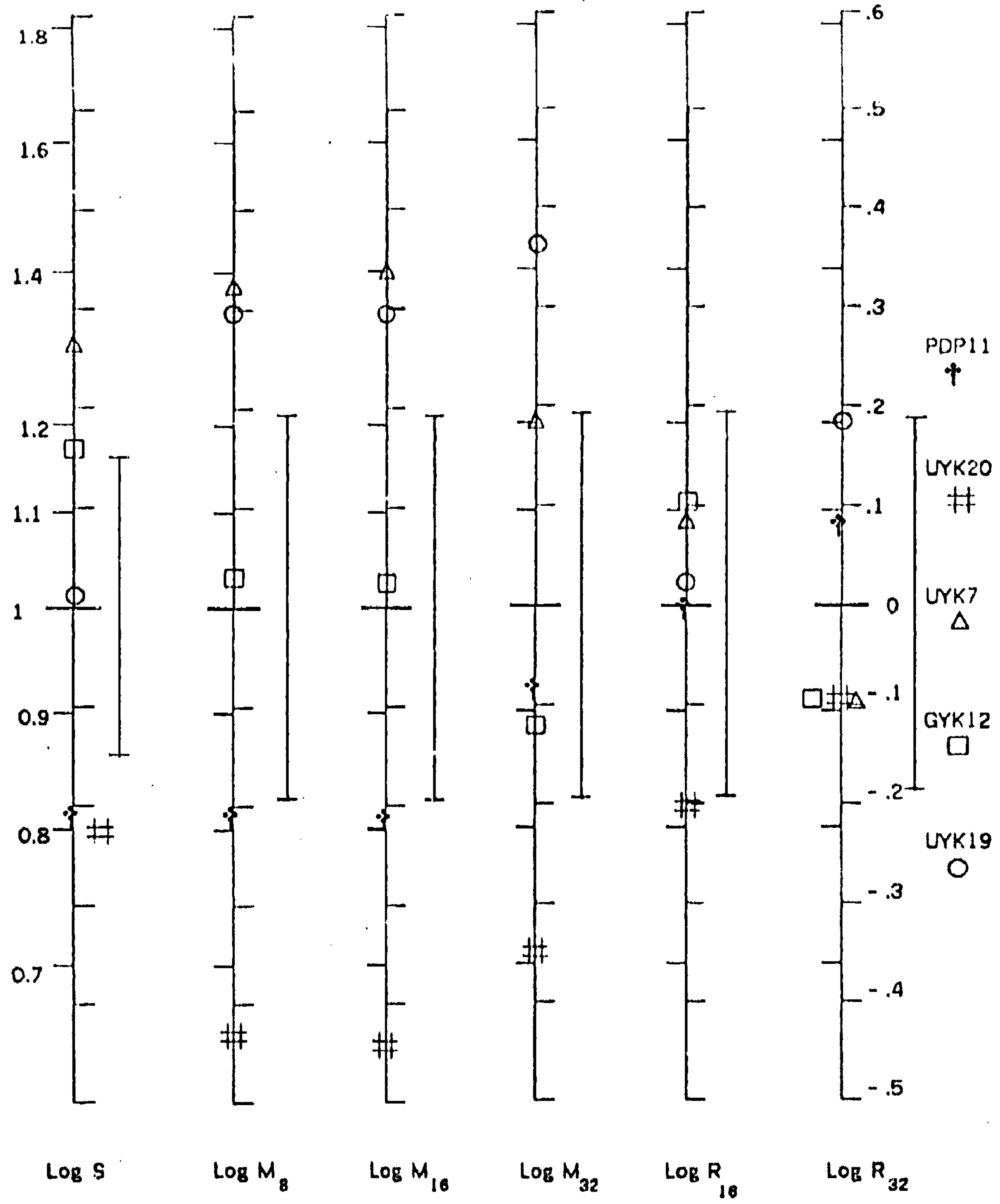
$\sigma_M^2 = (\Sigma_k\, M_k^2\ )\ /\ K-1$



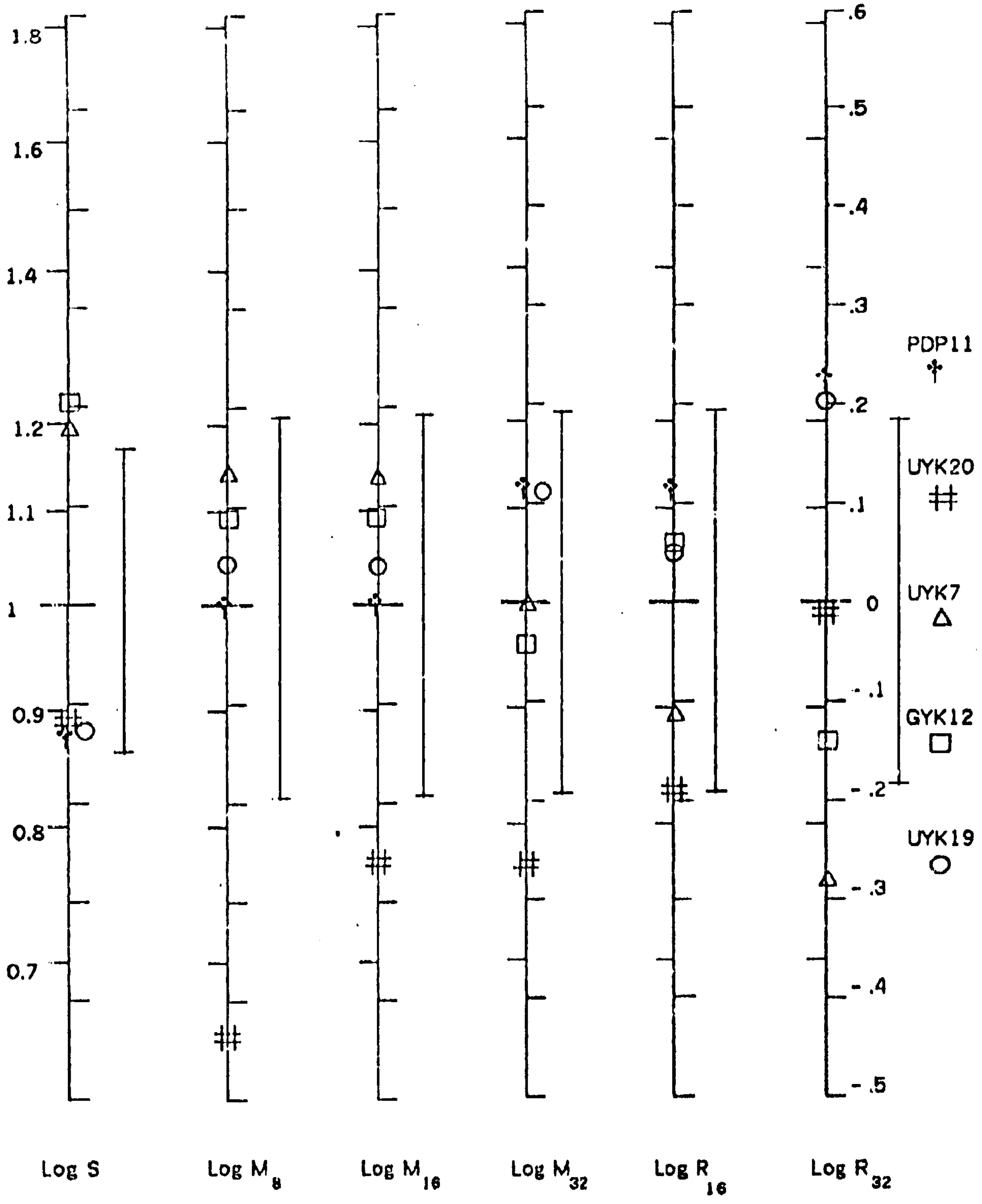
ALL TEST PROGRAMS
Confidence Intervals are Random 95%



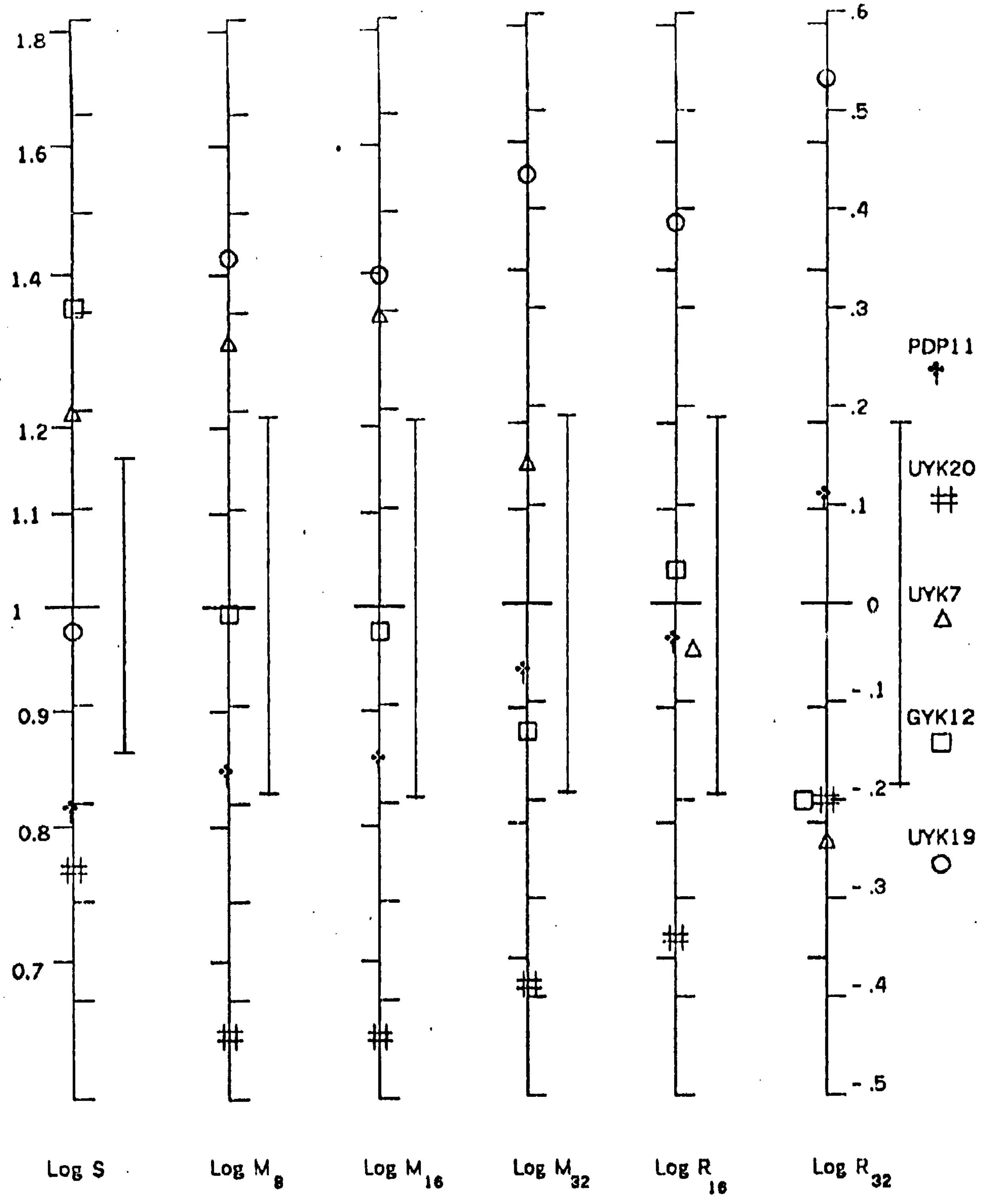
INTERRUPTS AND TRAPS
Confidence Intervals are Random 95%



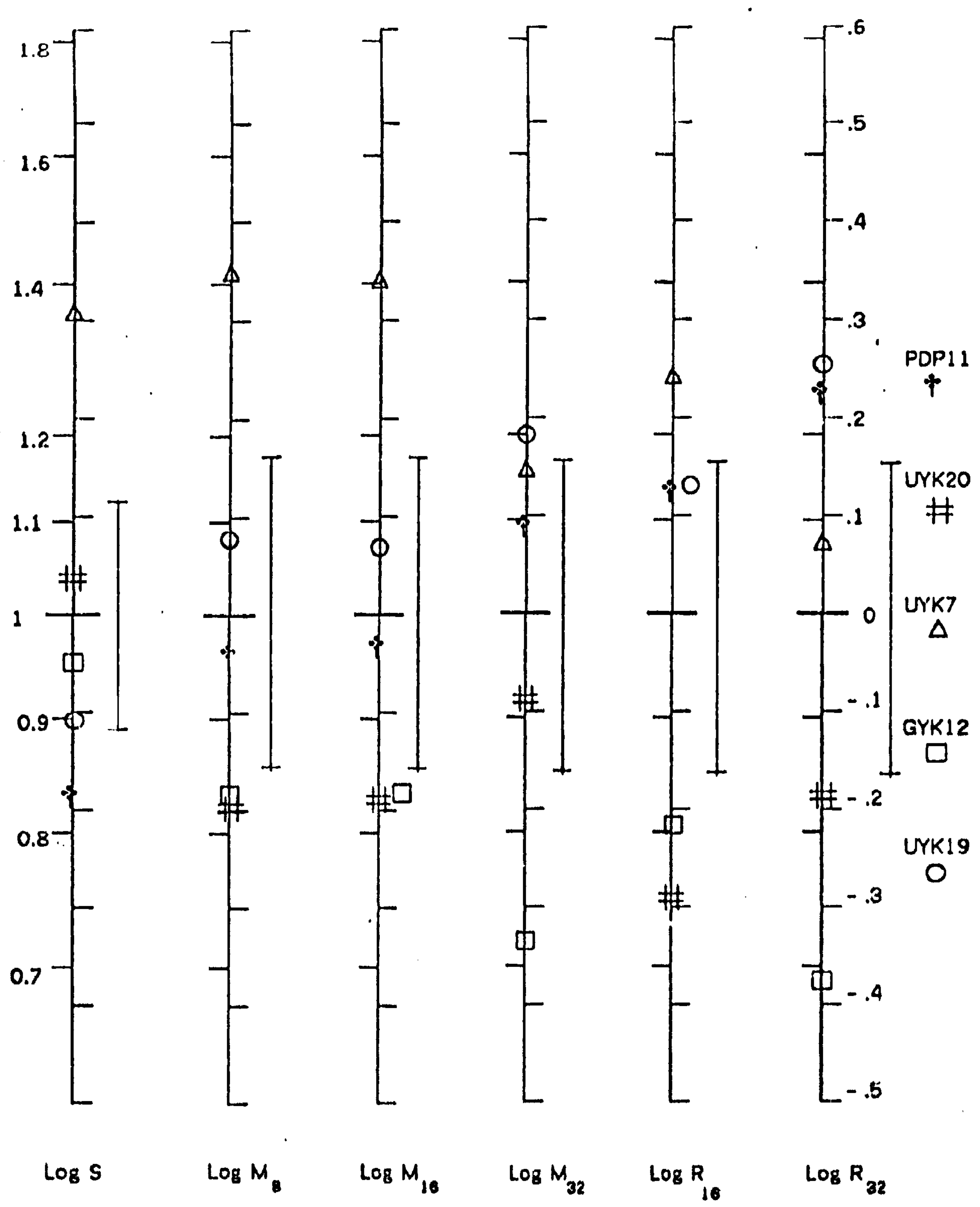
MISCELLANEOUS
Confidence Intervals are Random 95%



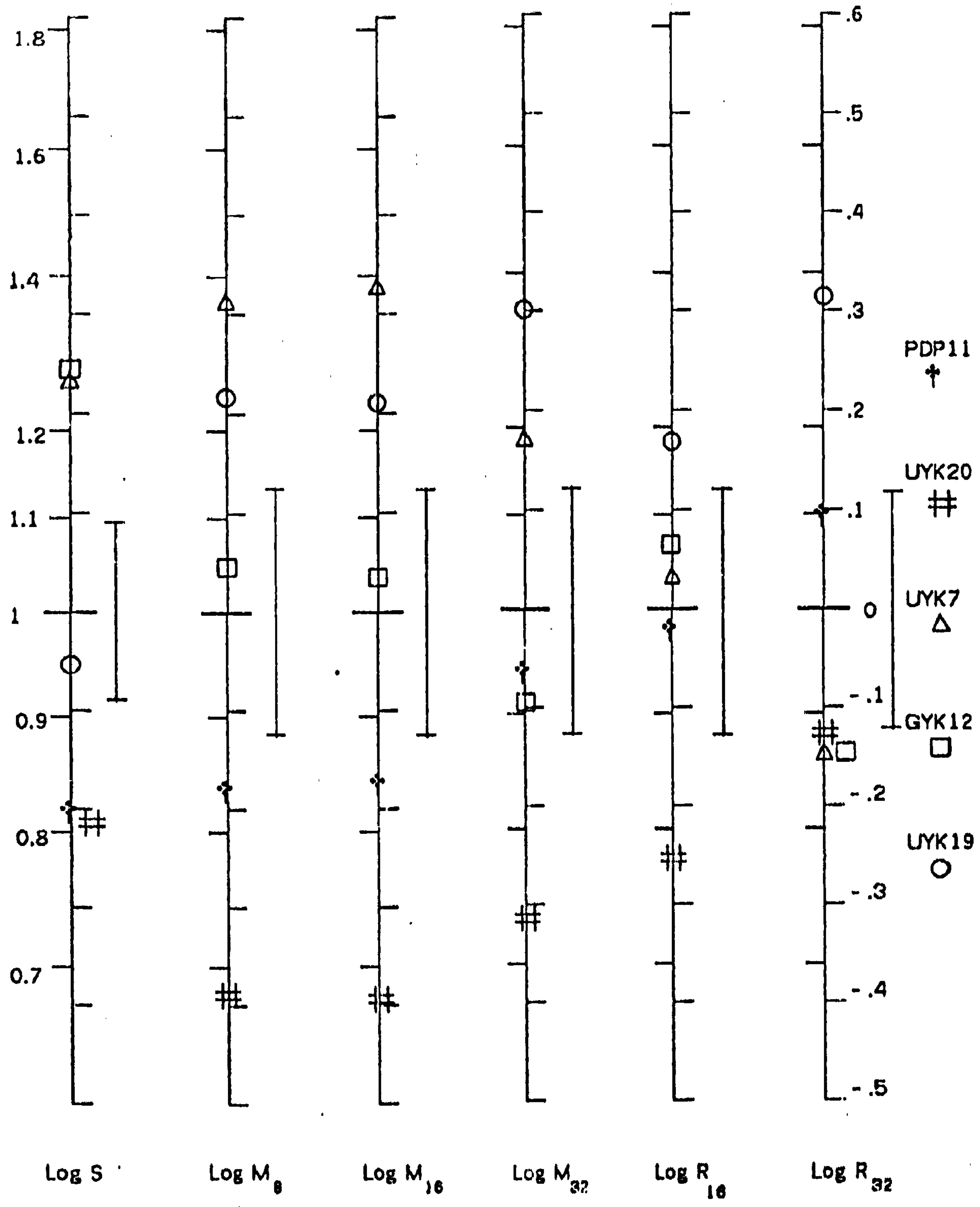
ADDRESS MANIPULATION
Confidence Intervals are R^ndom 95%



CHARACTER AND BIT MANIIPULATION
Confidence Intervals are Random 95%



EXECUTIVE MODE PROGRAMS
0 1 2 3 10 11



USER MODE PROGRAMS
4 5 6 7 8 9 12 13 14 15

APPENDIX 3 - PROGRAM MEASURES

	S					M(8)				
Machine	11	20	7	12	19	11	20	7	12	19
Prog/Pgmr										
0/7	94	164	228	148	130	2360	1088	7610	3524	4206
0/8	88	142	236	168	156	2294	1088	7568	3097	4052
1/5	96	162	246	164	98	252	354	572	434	220
1/10	118	160	304	184	120	350	418	884	576	326
2/6	214	734	472	264	238	620	1268	1312	528	502
2/9	202	440	428	192	232	460	872	1340	364	876
3/4	306	160	252	138	220	618	496	512	188	424
3/11	254	196	272	140	196	666	576	611	190	574
4/3	280	242	348	312	270	2598	1422	3144	1804	4010
4/12	240	174	256	316	258	2084	1064	1670	1716	2906
5/1	156	150	256	212	226	794	750	1656	964	1858
5/14	200	244	420	392	288	2018	1752	5396	4660	2978
6/2	274	298	376	360	374	2644	2848	5456	3288	4472
6/13	258	276	436	364	370	2610	2370	4592	3340	3778
7/0	54	68	176	116	64	960	966	2214	1840	1754
7/15	96	86	136	128	122	1510	1088	2414	2114	2622
8/0	88	102	172	152	94	898	818	1492	1246	838
8/15	120	136	180	212	114	1140	1022	2232	1708	986
9/2	144	178	196	256	122	220	278	396	344	230
9/13	156	132	204	192	132	282	210	372	304	256
10/1	224	202	248	244	226	2500	1376	1468	1992	2542
10/14	230	260	348	324	264	3042	1948	3864	3356	3226
11/3	250	292	320	352	300	11838	9100	7000	10196	14040
11/12	338	226	352	356	360	6880	4170	5892	5216	9832
12/4	90	116	162	580	140	762	992	2529	1636	2366
12/11	86	120	160	236	128	842	1370	3041	2668	2630
13/6	182	206	320	308	208	1490	882	2492	1896	1936
13/9	230	198	368	384	246	700	540	1520	1066	916
14/5	198	170	246	264	290	769	516	950	736	1312
14/10	348	204	302	170	294	2082	660	846	676	2488
15/7	278	256	444	392	282	4404	3666	5818	4962	7702
15/8	326	256	512	440	402	7046	5004	8442	5686	8236

	M[16]					M[32]				
Machine	11	20	7	12	19	11	20	7	12	19
Prog/Pgmr										
0/7	2512	1164	7782	3578	4206	3554	1528	7872	3904	6020
0/8	2370	1164	7700	3152	4052	3506	1530	7700	3482	5886
1/5	258	354	572	434	220	366	504	572	444	302
1/10	350	418	884	576	326	488	574	884	588	472
2/6	620	1268	1312	528	502	880	1638	1312	528	726
2/9	466	878	1340	364	876	632	1250	1340	416	1274
3/4	620	496	512	188	424	986	874	512	220	660
3/11	668	576	618	190	574	1014	974	620	216	972
4/3	2598	1422	3144	1804	4010	3370	1546	3144	1808	5342
4/12	2084	1064	1880	1716	2906	2550	1256	1880	1716	3760
5/1	794	750	1656	964	1858	1094	1068	1656	964	2250
5/14	2018	1752	5396	4660	2978	2634	2198	5396	4660	3554
6/2	2654	2848	5456	3288	4472	3595	3618	5456	3368	4884
6/13	2610	2370	4592	3340	3778	3364	2982	4592	3380	4118
7/0	960	966	2214	1840	1754	1428	1432	2236	1868	2202
7/15	1510	1088	2414	2114	2622	1948	1530	2416	2120	4300
8/0	902	818	1492	1246	838	1216	1028	1544	1324	1098
8/15	1140	1022	2232	1708	986	1548	1250	2232	1708	1372
9/2	220	278	396	344	230	286	370	396	364	294
9/13	282	210	372	304	256	376	288	372	304	344
10/1	2500	1376	1468	1992	2542	3202	1660	1468	1992	2986
10/14	3042	1948	3864	3356	3226	3696	2144	3864	3364	3694
11/3	11838	9100	7000	10196	14040	14819	11094	7000	10196	16220
11/12	6880	4170	5892	5216	9832	8998	5420	5892	5216	11666
12/4	832	1062	2808	1660	2364	1098	1340	2808	1732	3000
12/11	912	1440	3320	2668	2630	1218	1868	3320	2872	3602
13/6	1540	882	2492	1896	1936	1918	1044	2492	1996	2436
13/9	700	540	1664	1066	916	926	652	1664	1144	1242
14/5	782	522	968	736	1312	1008	626	968	760	1796
14/10	2088	660	864	676	2488	2370	784	864	700	2936
15/7	4404	3666	5818	4962	7702	5732	4644	5904	5172	9458
15/8	7046	5004	8442	5686	8236	8870	6234	8532	6100	10536

Machine	R(16)					R(32)				
	11	20	7	12	19	11	20	7	12	19
Prog/Pgmr										
0/7	2631	1009	6049	2281	4450	2631	1089	4238	1994	4450
0/8	1932	945	6282	2060	4360	1932	945	4255	1729	4360
1/5	237	249	311	313	246	237	249	274	237	246
1/10	322	292	550	385	367	322	292	401	290	367
2/6	547	1031	819	366	485	547	1031	638	280	485
2/9	496	694	800	252	960	496	694	603	221	960
3/4	559	130	611	168	328	543	130	509	125	328
3/11	750	179	659	171	334	734	179	538	130	334
4/3	4008	3261	7939	4055	5323	4008	3261	4509	3170	5323
4/12	4131	3195	3309	7109	4779	4131	3195	1983	3964	4779
5/1	2262	2069	2541	2346	2037	1558	1491	1606	1349	1795
5/14	4472	4207	7539	7834	4602	3766	3629	4692	4602	4360
6/2	4991	4526	4893	4229	4632	4049	4064	3401	2927	4257
6/13	5172	4034	4526	4347	4300	4102	3784	3116	2961	3897
7/0	1488	1203	1364	1651	2454	1488	1203	1175	1362	2454
7/15	2001	1314	1807	1974	1430	2001	1314	1476	1397	1430
8/0	1402	1086	1549	1326	1193	1402	1086	1059	1001	1193
8/15	1867	1688	3146	2331	1728	1867	1688	2008	1734	1728
9/2	212	238	223	291	308	212	238	205	210	308
9/13	290	207	225	244	311	290	207	205	175	311
10/1	2981	1956	1309	2254	2746	2981	1956	890	1606	2746
10/14	3893	2456	2074	3336	3371	3893	2456	1647	2264	3371
11/3	17995	14443	10095	15152	14367	16176	12848	6324	9758	14202
11/12	9389	7750	7868	8933	7723	7544	6155	4977	5511	7558
12/4	1169	1175	1647	1789	3051	1169	1175	1209	1263	3051
12/11	1149	1083	1706	2943	2843	1149	1083	1413	2153	2843
13/6	2277	1903	3882	3781	3200	2277	1903	2404	2233	3200
13/9	733	836	1084	1130	1059	733	836	815	838	1059
14/5	862	781	764	917	1640	862	781	558	657	1640
14/10	3685	886	738	851	3625	3685	886	534	566	3625
15/7	8308	6696	11827	11634	11751	8308	6512	7287	6939	11751
15/8	11397	6660	8207	6174	10173	11397	6476	5791	4414	10173